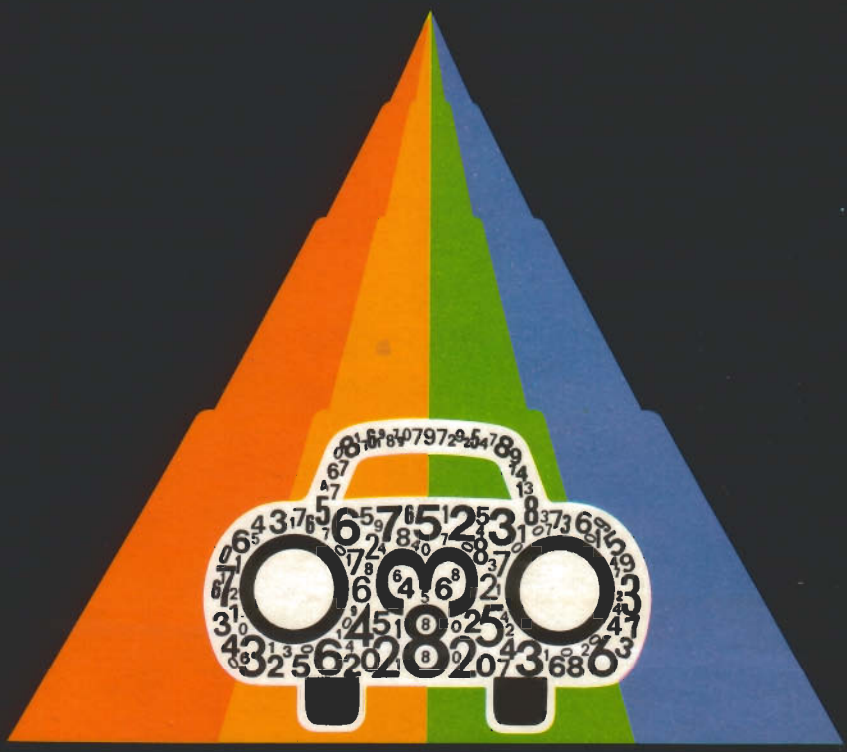
SPECTRUM INTRODUCCION AL CODIGO MAQUINA

cómo obtener más velocidad y potencia



IAN SINCLAIR

SPECTRUM INTRODUCCION AL CODIGO MAQUINA

Ian Sinclair

DIAZ DE SANTOS, S. A.
MADRID-ESPAÑA

Copyright © 1983 by Ian Sinclair
© Granada Publishing Limited
Título original: «Introducing Spectrum Machine Code»

Queda prohibida la reproducción total o parcial de la presente obra, bajo cualquiera de sus formas, gráfica o audiovisual, sin la autorización previa o escrita del editor, excepto citas en revistas, diarios o libros, siempre que se mencione la procedencia de las mismas.

ISBN: 0-246-12082-7 (edición en lengua inglesa).
ISBN: 84-86251-09-5 (edición en lengua española).

Depósito Legal: M-24024-1984
Impreso en Unigraf, S.A. Fuenlabrada, Madrid.

EDITA: DIAZ DE SANTOS, S. A.
C/. Lagasca, 95. MADRID-6
PRODUCCION: A.S.E.L., S. A.
C/. Clara del Rey, 41. Madrid-2
Telf.: 416 62 47
Traducido por: José Ignacio Soria
Revisado por: Aula de Informática Aplicada
C/. Conde de Peñalver, 38. MADRID-6

Contenido

Prólogo vii

1 La memoria: organización y tipos 1
2 El Spectrum por dentro 16
3 La unidad central de procesos 32
4 Descripción del Z-80 47
5 Operaciones con los registros 61
6 Diseño de programas 77
7 Entradas y salidas 92
8 Depuración y más programación 108
9 El final del recorrido 119
Apéndice A: Libros y revistas 137
Apéndice B: Números en coma flotante 138
Apéndice C: Codificación de las tablas 140
Apéndice D: Modos de direccionamiento del Z-80 141
Apéndice E: Tiempos de ejecución 142
Apéndice F: Códigos de operación del Z-80 143
Apéndice G: Códigos ASCII 155

Prólogo

Muchos usuarios de ordenadores se contentan con programar únicamente en BASIC. En cambio, muchos otros están deseosos de aprender sobre computadores más cosas que las que el BASIC puede proporcionarles.

Sin embargo, pocos de estos últimos parecen progresar mucho en el empleo del código máquina, y creo que esto se debe a que gran parte de los libros que tratan dicho código, suponen que los lectores ya están familiarizados con el vocabulario técnico y las ideas específicas del lenguaje máquina. Además, esos libros tienden a enfocar el estudio del código máquina de forma general, dejando al lector sin medio alguno de aplicar sus nuevos conocimientos a su propio computador.

Este libro tiene dos objetivos principales: uno es presentar al propietario de un Spectrum algunos detalles sobre el funcionamiento del ordenador, permitiendo, con ello, programar con mayor efectividad, incluso aunque no se profundice más en el estudio del código máquina. El segundo objetivo es presentar la potencia y la velocidad del lenguaje máquina, con ejemplos sencillos.

Debo recalcar la palabra "presentar". Ningún libro puede decirlo todo sobre el código máquina y a lo máximo que puedo aspirar, es a darle a usted, el lector, información suficiente para empezar. Empezar significa poder escribir rutinas cortas en lenguaje máquina, comprender los pequeños programas de este tipo que aparecen en las revistas y, por lo general, utilizar con mayor efectividad su Spectrum. También significa que usted podrá utilizar de manera eficaz libros sobre programación en código máquina, tales como los que se citan en el Apéndice A (estos libros son el punto de partida para trabajos mucho más avanzados). Desde ahí al dominio completo de la programación en lenguaje máquina, sólo hay un pequeño paso.

La comprensión del sistema operativo del computador y la capacidad de programar en el lenguaje de la máquina, pueden abrirle las puertas de un nuevo mundo dentro de los computadores. El conocimiento del sistema operativo le permitirá hacer cosas como renumerar

las líneas de un programa, transformar las instrucciones PRINT en LPRINT con un solo comando, alterar el sonido "click" que produce el teclado, o imprimir una lista de todas las variables.

Los programas en código máquina le proporcionarán un control absoluto sobre el ordenador, de forma que podrá realizar tareas como leer cintas del ZX-81, dirigir una impresora (con conexión serie) a través del puerto del magnetofón, acelerar acciones como conversión decimal/hexadecimal o los gráficos de pantalla, etc... Tengo que insistir en que este libro no es un libro de programas, sino de explicaciones, puesto que únicamente aprenderá a programar eficientemente en lenguaje máquina "haciendo sus propios programas". Ningún operario puede trabajar sin buenas herramientas. Las herramientas que yo he utilizado han sido: el Spectrum, un magnetofón de cassettes TROPHY CR 100, un receptor de televisión PHILIPS 14CT3005 y la impresora ZX. En cuanto a herramientas "software" (programas), he empleado: el desensamblador "dpas" de Campbell Software, que me ha permitido investigar en el sistema operativo y, después, el ensamblador "ULTRAVIOLET", de ACS Software, que ha hecho mucho más rápida y sencilla la codificación de programas escritos en lenguaje ensamblador. Finalmente, debo dar las gracias a toda la gente que hizo posible este libro: Richard Miles de Granada Publishing, por darme ánimos, Bill Nichols y Jane Boothroyd de Investigaciones Sinclair, por prestarme un Spectrum de 16 K, a mi esposa por su paciencia y a la señorita Leake, por su hospitalidad. Si añade a esta lista los nombres de Hector Berlioz y Jean Sibelius, por darme serenidad, tendrá el equipo completo que ayudó al escritor. Tal vez debería mencionar, también, el hecho de que yo no tengo ninguna relación ni con Clive Sinclair, ni con investigaciones Sinclair. El sistema operativo del Spectrum tiene protegidos los derechos de copia (copyright), y las direcciones de dicho sistema, que aquí aparecen y que no se citan en el manual, no fueron proporcionadas ni aprobadas por Investigaciones Sinclair.

Ian Sinclair

Capítulo 1

La memoria: organización y tipos

Una de las cosas más desalentadoras a la hora de estudiar el nivel que existe debajo de BASIC, es la cantidad de palabras técnicas específicas (el argot), que encontramos. Los autores de muchos libros sobre computadores suponen que los posibles lectores tienen ya ciertos conocimientos de electrónica, de forma que, si realmente esto es verdad, no hay ningún problema para avanzar a través de dichos libros. Yo partiré de la base de que usted no posee ninguna formación en electrónica, y todo lo que puedo pedirle es una cierta experiencia en el empleo de BASIC en el Spectrum, preferiblemente, escribiendo sus propios programas en BASIC. Empezaremos, pues, en el sitio correcto: al principio de todo. Como no deseo interrumpir a lo largo del libro las explicaciones con la inclusión de detalles matemáticos y técnicos, cuando se necesitan esos detalles, remito al lector a los Apéndices correspondientes, de manera que éstos pueden consultarse o no, según convenga en cada momento. Al principio de todo, tenemos la memoria. La unidad de memoria, en lo que respecta a nosotros, es un circuito electrónico que funciona como un interruptor.

Cuando entramos en una habitación y encendemos la luz, nunca pensamos que el hecho de que ésta permanezca encendida hasta que la apaguemos tenga nada de especial. Nunca le diremos a nuestros amigos, con tono trascendental, que el circuito de la luz posee una memoria; y, sin embargo, cada unidad de memoria del computador no es más que un tipo de interruptor muy pequeño, que puede encenderse o apagarse y permanecer en el estado en que se encuentra, hasta que sea utilizado de nuevo. Esta unidad elemental de la memoria se denomina un *bit* (el nombre es la contracción de "binary digit", que significa dígito binario en inglés). Vamos a conservar el modelo del interruptor, ya que es muy útil. Suponga que queremos emplear los interruptores y circuitos eléctricos para representar información. Para ello, podemos construir un circuito como el de la Figura 1.1. Cuando el interruptor se enciende, también lo hace la bombilla; podemos establecer que este estado significará "SI". Si apagamos el interruptor, la bombilla se

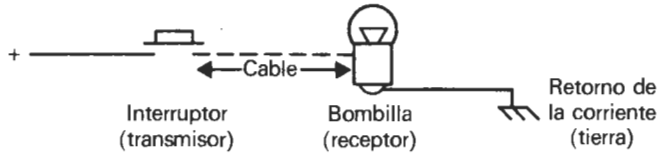
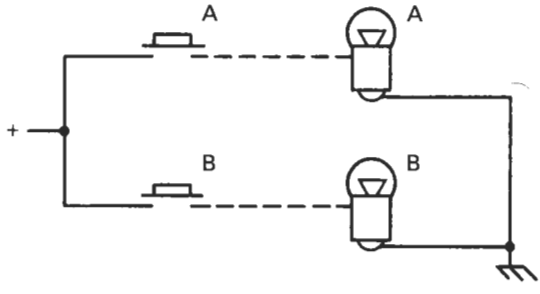


Fig. 1.1. Sistema de señalización de línea única con interruptor y bombilla.

apaga; a este nuevo estado le podemos atribuir el significado de "NO". Podríamos asignar otros significados cualesquiera a los estados, siempre que sólo fuesen dos.

Las cosas funcionarán mejor si tenemos dos interruptores, dos bombillas y dos líneas, como las de la Figura 1.2. Ahora, existen cuatro combinaciones posibles: (a) ambos interruptores apagados; (b) A encendido y B apagado; (c) A apagado y B encendido; (d) los dos interruptores encendidos.



A	B
Apagado	Apagado
Apagado	Encendido
Encendido	Apagado
Encendido	Encendido

Fig. 1.2. Señalización con 2 líneas: ahora pueden enviarse hasta 4 señales distintas.

Esto significa que podríamos atribuir cuatro significados diferentes, uno a cada estado posible. Si utilizamos una línea, tenemos dos significados; con dos líneas, cuatro significados ($2 \times 2 = 4$); y, si desea averiguarlo, encontrará que, si se utilizan tres líneas, tendremos ocho estados posibles, a los que podemos dar ocho significados diferentes. Puesto que 8 es igual a $2 \times 2 \times 2$, no debería ser muy difícil adivinar que ocho líneas nos permitirían $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$ significados distintos, representado cada uno por un estado del circuito total.

Realmente, para N líneas el número de estados posibles es de 2^N. Cualquier conjunto de ocho interruptores, cada uno de los cuales puede estar encendido o apagado, puede adoptar hasta 256 configuraciones diferentes. Es asunto nuestro darle algún sentido a esas configuraciones. Un modo especialmente útil de utilizar los estados, es el denominado código binario. El código binario es una forma de representar los números por medio de dos únicos dígitos: Ø y 1. A menudo, el cero se representa atravesado por una barra, para distinguirlo de la letra O. Podemos asociar el cero con el estado "apagado" y el uno con el estado "encendido", y esto nos permitiría representar 256 números distintos con los ocho interruptores, pensando en las posiciones de los mismos como si fuesen dígitos: Ø para apagado y 1 para encendido. Este grupo de ocho se llama un *byte* (un octeto), y por esta razón aparece tantas veces el número 256 en el mundo de los ordenadores. ¿Por qué un grupo de ocho?, se preguntará. Bien, simplemente las cosas se desarrollaron de ese modo: las primeras calculadoras podían trabajar con cuatro bits a la vez; después se pasó a los ocho bits, y eso ha permanecido vigente mucho tiempo. Las máquinas de 16 bits aún no son muy comunes. La forma de colocar los bits dentro de un octeto para que éste represente un número, sigue las mismas reglas que empleamos nosotros para escribir números normalmente. Cuando se escribe un número como 256, por ejemplo, el 6 significa 6 unidades, el 5 se escribe inmediatamente a la izquierda e indica el número de decenas, y el dos, de nuevo a la izquierda del anterior, indica las centenas. Estas *posiciones* nos dan la importancia o el *peso* de un dígito (ver figura 1.3.). El 6 de 256 se denomina "dígito menos significativo", mientras que el 2 es el "dígito más significativo". Si cambiamos el 6 por un 7, obtenemos una variación de una unidad en 256. Por el contrario, si substituimos el 2 por un 3, lo que resulta es un cambio de 100 unidades en 256, mucho más importante.

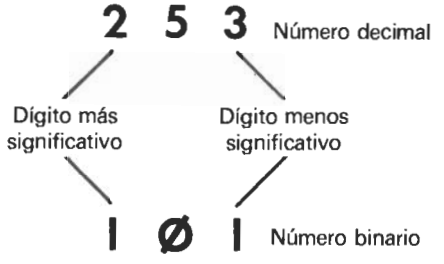


Fig. 1.3. Significado de los dígitos. En nuestro sistema de numeración, a diferencia del Romano, se usa la posición de los dígitos para indicar su importancia (peso).

Una vez que hemos visto brevemente los bits y los octetos, es hora de volver un momento al modelo de la memoria como conjunto de interruptores. Sucede que necesitamos dos tipos de memoria. Un tipo debe ser permanente, como los interruptores mecánicos o las conexiones fijas, porque esta memoria se va a emplear para almacenar las instrucciones codificadas con números que controlarán al computador. A esta clase de memoria se la denomina ROM, palabra que procede de "Read Only Memory", que significa "memoria sólo de lectura", en inglés. La ROM es la parte más importante de su ordenador, ya que contiene las instrucciones que permiten que aquel lleve a cabo todas sus acciones. Cuando usted escribe un programa propio, almacena otro conjunto de instrucciones codificadas numéricamente en otra parte de la memoria y, naturalmente, querrá que esa otra parte pueda ser utilizada una y otra vez. Esta es una clase diferente de memoria en la que se puede "escribir" y de la que se puede "leer", y, si fuésemos consecuentes, debería llamarse "memoria de lectura y escritura" (RWM, en inglés). Desafortunadamente, aquí no se aplica la lógica y se suele denominar RAM (Random Access Memory), que son las iniciales inglesas de "memoria de acceso aleatorio", porque éste era un nombre utilizado en los primeros tiempos de los ordenadores para distinguir este tipo de memoria de otro que funcionaba de forma distinta. El nombre de RAM se ha mantenido hasta nuestro días, y tendremos que usarlo del mejor modo posible.

Todo hecho con números

Volvamos de nuevo a los octetos. Ya hemos visto que un octeto, que es un grupo de ocho bits, puede tener hasta 256 configuraciones diferentes de los bits que lo forman, y que lo más útil es emplear estas configuraciones para representar un número con cada una de ellas, en el llamado código binario. Los números serán los comprendidos entre Ø y 255 (no de 1 a 256, pues necesitamos un código para el cero), y cada octeto de los 16.384 octetos que componen la RAM del Spectrum de 16 k, puede almacenar un número de ese intervalo. Los números no dicen nada por sí mismos, y si un computador sólo pudiese manejar números del Ø a 255, no valdría para hacer gran cosa. Por este motivo, los números se emplean como *códigos*. Del mismo modo que una tecla del Spectrum sirve para realizar diferentes acciones, cada código numérico puede utilizarse con diferentes significados. Si usted ya ha programado en BASIC, sabrá que cada letra del alfabeto, cada dígito del Ø al 9, y cada signo de puntuación, está codificado con un número entre 32 (que es el código del espacio) y 127 (que es el signo "copyright" del

Spectrum). Eso deja una gran cantidad de códigos numéricos libres para otros propósitos como codificar los caracteres gráficos o, al igual que la mayoría de los pequeños ordenadores, para codificar acciones específicas.

Cuando, por ejemplo, se pulsa la tecla etiquetada con PRINT, lo que se almacena en la RAM de su Spectrum no es la secuencia de cinco octetos correspondientes a los códigos ASCII de las letras PRINT, que serían 80, 82, 73, 78 y 84, sino un único octeto: 245. Este único octeto se denomina "token", y el computador puede usarlo para dos cosas diferentes. Por un lado, sirve para localizar los caracteres que componen en realidad la palabra PRINT. Estos se encuentran guardados, como códigos ASCII numéricos, en la memoria ROM, puesto que no cambian nunca (¡a usted no le gustaría que apareciese la palabra PERPLEJO cuando pulsase la tecla PRINT!), gracias a lo cual no ocupan sitio en la RAM. La otra misión de un token es localizar un conjunto de instrucciones (también codificadas numéricamente en la ROM) que llevarán a cabo la acción de representar algo en la pantalla. Estas últimas instrucciones con forma de códigos numéricos, constituyen el "código máquina", denominado así porque controla lo que la máquina hace en todo momento.

Un intermedio práctico

Para ayudarle a que asimile los conceptos anteriores, vamos a probar un corto programa, el de la Figura 1.4., que está diseñado para descubrir qué "palabras clave" están almacenadas en la ROM. El programa utiliza la instrucción PEEK de BASIC, que debe ir seguida por un número o una variable numérica y cuyo significado es: "averigua qué octeto está almacenado en esa dirección". Los grupos de ocho bits (octetos) en que se divide la memoria del Spectrum, están numerados desde Ø en adelante, de manera que cada octeto tiene un número distinto, pertenezca a la memoria ROM o a memoria RAM. Esto es muy parecido a cómo se numeran las casas que hay en una calle, y por eso nos referimos a los números de los octetos como sus "*direcciones*".

```
1Ø PRINT 15Ø;"        ";: FOR n=15Ø TO 516
2Ø LET k=PEEK n
3Ø IF k <=127 THEN PRINT CHR$ k;
4Ø IF k >=128 THEN PRINT CHR$ (k-128) :
   PRINT n;"        ";
5Ø NEXT n
```

Fig. 1.4. Programa BASIC para ver cómo están almacenadas en la ROM las palabras clave.

La función PEEK averigua qué número se encuentra almacenado en una dirección de memoria dada (el valor del número debe estar entre Ø y 255) y el Spectrum convierte automáticamente los números codificados en binario (que es como están en la memoria) en números normales (cuyo nombre correcto es *decimales)*. Si utilizamos la función CHR$, podemos imprimir el carácter cuyo código es el número que hemos obtenido con PEEK. Hasta ahora, todo va bien. El programa emplea "n" como número de dirección, y después, comprueba que PEEK n es menor que 128, en otras palabras, que es un carácter en código ASCII. Si lo es, se imprime en la pantalla. La necesidad de esta comprobación se debe a que el último carácter de cada grupo de palabras o de cada palabra clave, se encuentra guardado con una codificación distinta de la normal. El número recuperado por PEEK para el último carácter, es 128 + el código ASCII de dicho carácter, en vez del código ASCII habitual. Por ejemplo, las tres primeras posiciones que examina nuestro programa (mediante PEEK), son las direcciones 15Ø, 151 y 152, que contienen los números 82, 79 y 196. El número 82 es el código ASCII de R, 79 es el de N y 196 − 128 = 68, que sería el código ASCII de D; por lo tanto, en estas tres posiciones estaría RND almacenada. ¿Para qué complicarse cambiando el código de la D? La razón es que los que diseñaron el Spectrum no querían ocupar mucha memoria; por ello, en lugar de emplear un cuarto de octeto para separar RND de la palabra clave siguiente, que es INKEY$, inventaron esta forma de decir al computador dónde termina cada palabra clave. Cuando el Spectrum lee estos códigos de uno en uno, está programado para dejar de leer cuando llega a uno que es mayor que 128. Nosotros hemos hecho lo mismo en la línea 4Ø del programa de la figura 1.4., para imprimir la letra correcta (restando del código 128 antes de usar CHR$), y también incluimos varios espacios antes de pasar a la letra siguiente.

Vamos ahora con la siguiente revelación. Dele un vistazo a la tabla de palabras clave que aparece en el manual de su Spectrum. Observe que están listadas en el mismo orden en que se encuentran almacenadas en la memoria. Gracias a que existe este orden de almacenamiento, con los códigos de "token" correspondientes (que empiezan en 165) en el mismo orden, es sencillo para la máquina encontrar un código, si se le proporciona la dirección de comienzo de la lista (que es lo que sucede cuando usted pulsa una tecla).

Análisis del Spectrum

En la figura 1.5. aparece un diagrama del Spectrum. Es bastante simple, porque he omitido en él todos los detalles; pero es suficiente

para proporcionar una pista de cómo funcionan las cosas. A esta clase de diagramas se la denomina "diagrama de bloques", debido a que cada unidad está dibujada como un bloque, sin especificar detalles de lo que pueda haber en el interior de dicha unidad.

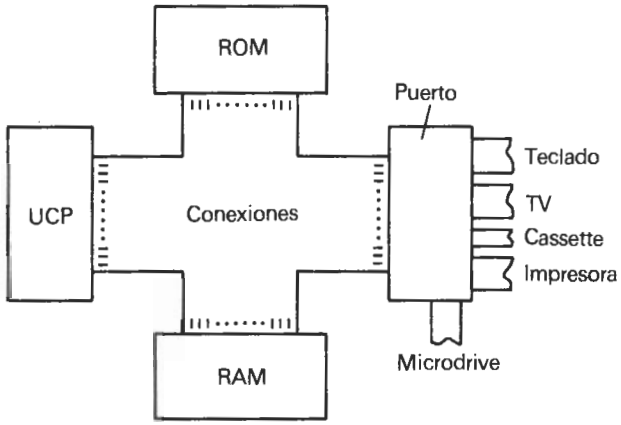


Fig. 1.5. Diagrama de bloques del Spectrum. Las conexiones consisten en un gran número de líneas eléctricas que unen todas las unidades del sistema.

Los diagramas de bloques son parecidos a los mapas a gran escala, que nos muestran las carreteras nacionales que van de ciudad en ciudad, pero no las carreteras comarcales o las calles de las ciudades. Uno de tales diagramas es suficiente para enseñarnos los caminos principales que siguen las señales dentro del computador, evitando las confusas especificaciones que serían necesarias para describir con exactitud las conexiones eléctricas existentes en la máquina. Dos de los bloques del diagrama ya han sido presentados: las memorias ROM y RAM. La ROM es la memoria que no permite la escritura en ella; contiene todas las instrucciones esenciales (además de las palabras claves y los números de tokens), que se precisan para hacer funcionar el computador. La memoria RAM sirve para guardar sus programas y otras muchas cosas que veremos más tarde.

El bloque etiquetado con "U.C.P." es fundamental. U.C.P. significa Unidad Central de Proceso (en otros diagramas puede aparecer como Unidad de Microprocesador), y es el "cerebro" del sistema. La palabra "Unidad" está bien escogida en este caso, puesto que la UCP es un único circuito integrado, uno de esos "chips" de silicio sobre los que habrá leído algo, encapsulado en una plancha de plástico negro y provisto de 40 "patas" (pins es el nombre inglés) dispuestas en dos filas de 2Ø, como muestra la figura 1.6. Existen diferentes tipos de UCP

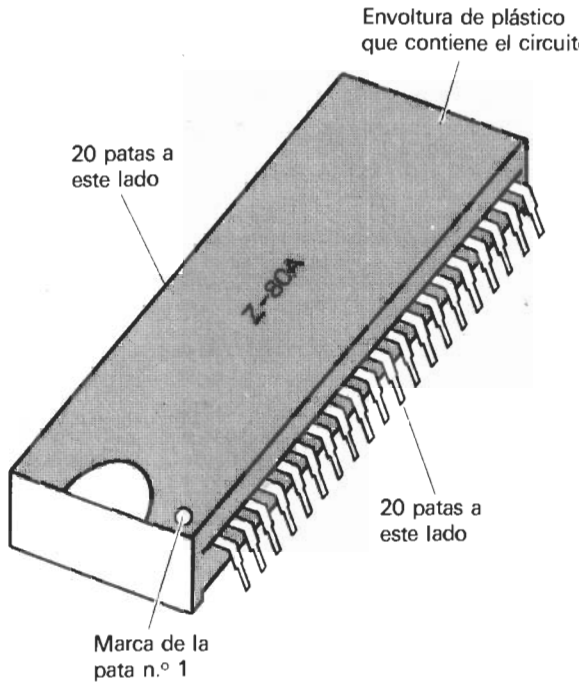


Fig. 1.6. La UCP Z-80. El circuito activo es, en realidad, más pequeño que una uña, y la envoltura de plástico externa (52 mm. × 14 mm.) hace que sea más fácil su manejo.

construidas por distintos fabricantes, y la que lleva el Spectrum se llama Z-80 A. Este modelo es casi idéntico al denominado Z-80, diferenciándose únicamente en que el Z-80 A puede trabajar a mayor velocidad si es necesario. ¿Qué hace la UCP? La respuesta es que lo hace prácticamente todo, y a pesar de ello, sólo puede llevar a cabo muy pocas acciones y muy simples. La UCP puede *cargar* un octeto, lo que significa que cualquier octeto almacenado en la memoria puede copiarse en otra memoria existente dentro de la UCP; otra cosa que puede realizar es *almacenar* un octeto, o sea, un octeto guardado en la UCP, puede copiarse en cualquier dirección de la memoria RAM.

Ambas acciones (figura 1.7.) son las que se ejecutarán más frecuentemente durante el tiempo de vida de la UCP, y, combinándolas, podemos copiar un octeto de cualquier dirección, en cualquier otra dirección de memoria. ¿No parece esto muy útil?: pues resulta que eso es, precisamente, lo que ocurre cuando usted pulsa la tecla "j" y ve aparecer dicha letra en la pantalla. La UCP trata el teclado y la pantalla como si fuesen una parte de la memoria, y traslada los octetos de uno a otro cuando usted teclea. Esta es una considerable simplificación del proceso, pero servirá muy bien por ahora.

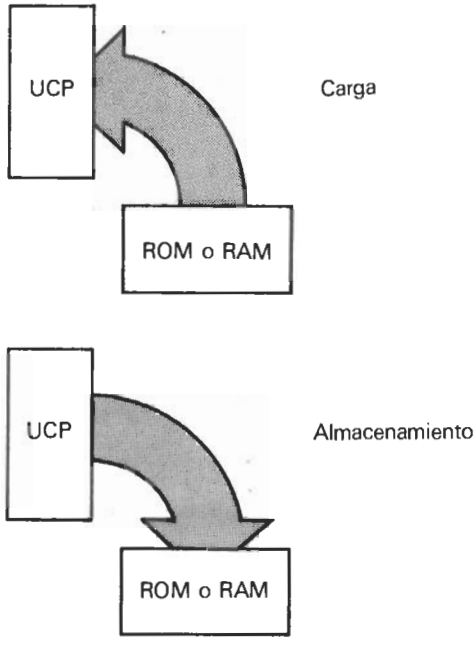


Fig. 1.7. Operaciones de carga y almacenamiento.

La carga y el almacenamiento son dos operaciones muy importantes de la UCP, pero hay otras. Entre ellas, está el conjunto de operaciones aritméticas. Contrariamente a lo que se podría esperar, consisten únicamente en la suma y la resta, y con números de dos octetos como máximo. Quizá se pregunte cómo ejecuta el computador operaciones aritméticas con números más grandes o con números fraccionarios; o cómo efectúa las divisiones, exponenciaciones, logaritmos, cálculo de senos y cosenos, etc. Pues bien: por medio de programas en código máquina, contenidos en la memoria ROM. Si no existiesen esos programas, tendría usted que escribir los suyos, y un programa en BASIC para multiplicar números, que sólo emplease la suma, sería largo y aburrido: ¡no es una perspectiva agradable! Existe también un conjunto de instrucciones lógicas. La lógica de la UCP es, como todas sus acciones, muy sencilla, y obedece a reglas muy estrictas. Las instrucciones lógicas comparan los bits de dos octetos, y producen un *resultado* que depende de los valores de los bits comparados (Ø ó 1) y de la regla lógica que se use en cada caso. Las tres reglas lógicas posibles se llaman AND (y), OR (o-incluso) y XOR (o-exclusivo). La figura 1.8. muestra su funcionamiento.

Otro conjunto de instrucciones es el llamado juego de instrucciones de salto. Un salto significa un cambio de dirección, muy similar a la

instrucción GOTO de BASIC, y sirve para que la UCP tome decisiones en ciertos momentos. De la misma forma que se puede programar en BASIC:

```
1ØØ IF a=36 THEN GOTO 1Ø5Ø
```

se puede lograr que la UCP ejecute una instrucción en una dirección completamente diferente de lo normal, que sería la dirección siguiente a la actual. La UCP es un dispositivo programable, lo que quiere decir que ejecuta cada acción como resultado de haber recibido antes una instrucción que estaba almacenada en un octeto de la memoria. Por lo general, cuando la UCP recibe una instrucción, que está en una dirección en alguna parte de la memoria (normalmente en la ROM), ejecuta dicha instrucción y después "*lee*" la instrucción almacenada en la dirección de memoria inmediatamente superior a la anterior. Una instrucción de salto evitaría que ocurriese eso, y forzaría a la UCP a leer la instrucción siguiente en otra dirección, que estará especificada en la propia instrucción de salto. Se pueden hacer saltos dependientes del resultado de acciones anteriores, como por ejemplo, del resultado cero, positivo o negativo, de una resta, suma o comparación.

AND (y)

El resultado de la operación AND entre dos bits, será 1, si ambos bits son igual a 1; Ø en otro caso:

```
1 AND 1=1     1 AND Ø=Ø     Ø AND Ø=Ø
              Ø AND 1=Ø
```

Para dos octetos, se hace un AND entre cada par de bits correspondientes:

```
        1Ø11Ø111
AND     ØØØØ1111
        ØØØØØ111
```

Pues solo
esos bits
son 1 en *ambos*
octetos

OR (o-inclusivo)

El resultado de la operación OR entre dos bits será 1, si uno de ellos, o ambos, son igual a 1; Ø en otro caso:

1 OR 1 = 1 1 OR Ø = 1 Ø OR Ø = Ø
Ø OR 1 = 1

Para dos octetos, se hace un OR de cada uno de los pares de bits correspondientes:

```
     1Ø11Ø111
OR   ØØØØ1111
     --------
     1Ø111111
```

Pues es el único
bit que es Ø
en ambos
octetos

XOR (o-exclusivo)

Es como OR, pero el resultado es cero si ambos bits tienen el mismo valor:

1 XOR 1 = Ø { 1 XOR Ø = 1 } Ø XOR Ø = Ø
  { Ø XOR 1 = 1 }

```
     1Ø11Ø111
XOR  ØØØØ1111
     --------
     1Ø111ØØØ
```

Si dos bits
son idénticos,
el resultado
es cero

Fig. 1.8. Reglas de las operaciones lógicas AND, OR y XOR.

Esta no es la lista completa de acciones de la UCP, pero las instrucciones que no he mencionado no son muy importantes, ni tampoco muy distintas de las expuestas. Lo que quiero resaltar es que

el "mágico microprocesador" no es un circuito tan inteligente. Lo que hace tan importante al computador es que puede ejecutar sus instrucciones con mucha rapidez y cada una de sus acciones se controla con un programa, enviando al ordenador señales eléctricas.

Estas señales se envían a ocho patas llamadas "pins de datos", de la UCP y, como habrá supuesto, esos ocho "pins" corresponden a los ocho bits de un octeto codificado en binario. Cada octeto de la memoria, por tanto, podrá afectar al microprocesador, compartiendo sus señales eléctricas con la UCP. Una descripción en tales términos es muy larga para escribirla más de una vez; por ello, hablaremos de lecturas y escrituras, siempre desde el punto de vista de la UCP, de tal modo que un bit 1 provoca una señal "1" en el pin de datos, y un bit Ø provoca una señal "Ø", en el correspondiente pin de datos (no debe olvidar que los "pins" son las "patitas" del circuito integrado, en este caso de la UCP).

Al igual que la lectura de un papel o la audición de una cinta musical no destruyen lo que hay escrito o grabado en ellos, la lectura de la memoria no cambia el contenido de ésta de ningún modo, permanece inmutable. El proceso opuesto, la escritura, sí que lo hace, y el contenido de la memoria cambia. En la escritura, como cuando grabamos una cinta magnetofónica, se borrará cualquier cosa que existiese anteriormente, y cuando la UCP escribe un octeto en una dirección de la memoria, lo que estuviese allí previamente desaparece y es reemplazado por el nuevo octeto. Esta es la razón de que sea tan fácil escribir en BASIC nuevas líneas que reemplacen a algunas ya existentes, simplemente poniendo en las nuevas los mismos números que poseían las antiguas.

Basic y código máquina

¿Realmente se escriben programas en BASIC? Esta puede parecer una pregunta tonta, pero es muy seria. Todo lo que hacen los programas, es ejecutado por una serie de instrucciones, codificaciones de manera adecuada para la UCP y, hasta ahora, usted nunca ha escrito ninguna de esas instrucciones. Lo único que hace es elegir dentro de una gama de posibilidades, que llamamos las palabras clave de BASIC, y ordenarlas de la forma que le parece apropiada para obtener los resultados correctos. Nuestra elección está limitada a las palabras clave contenidas en el interior de la memoria ROM del computador. Como no podemos cambiar la ROM, si deseamos ejecutar acciones distintas a las que ésta proporciona, tenemos dos caminos: o bien intentamos combinar comandos BASIC, o trabajamos directamente con el código

máquina de la UCP. Es algo así como la diferencia entre decir "vehículo motorizado con capacidad para transportar más de ocho personas" y decir "autobús". Cuando queremos llevar a cabo acciones con sólo un número limitado de comandos, el resultado puede ser algo primitivo y poco brillante, especialmente si cada comando es, a su vez, un conjunto de otros comandos. La acción directa es rápida, pero puede ser difícil. Esta "acción directa" a la que me refiero es el código de máquina, y una gran parte de este libro está dedicada a comprender este "lenguaje", ¡que es complicado, porque es muy simple!

Veamos una situación que ilustre esta paradoja. Suponga que quiere tener un muro en la parte posterior del jardín de su casa. Podría contratar a un albañil y lo único que tendría que hacer es decirle dónde desea que se contruya el muro, y luego, sentarse a esperar. Eso sería el equivalente de emplear BASIC con una palabra clave que fuese "construye un muro". Hay que hacer muchas cosas, pero usted no se preocupa de los detalles de la construcción. Piense ahora en otra posibilidad: posee un robot que ejecuta órdenes sin pensar, pero con una rapidez increíble. Ahora no podría decirle al robot que construya un muro, porque ese tipo de instrucciones está más allá de su entendimiento. Tendrá que decirle con todo detalle qué debe hacer exactamente en cada paso, es decir: "traza una línea desde un punto que está a 30 centímetros de la pared de la cocina y junto a la valla, hasta otro punto a 30 centímetros de la pared del salón y junto a la valla opuesta. Mezcla tres sacos de arena y dos de cemento con cuatro carretillas de guijarros. Luego, mézclalo todo con agua y llena con ello un cubo, que se vacía poniéndolo bocabajo. Cava una zanja a lo largo de la línea que trazaste antes y rellena dicha zanja con la mezcla del cubo..." Las instrucciones son muy específicas, puesto que se dan a un robot sin cerebro, pero se cumplirán exactamente y con una velocidad enorme. Si se olvida de decirle al robot cualquier cosa, ésta no se realizará por muy obvia que le parezca. De esa forma, si no mencionase cuánta masa, qué proporción de componentes lleva la mezcla o dónde hay que usarla, acabará teniendo los ladrillos unos encima de otros sin masa que los sujete. Si, en cambio, no se acuerda de decirle a la máquina la altura del muro, así como el número de niveles de ladrillos necesarios, el robot comenzará a poner filas y filas de éstos, unas encima de otras (al estilo de la película "El Aprendiz de Brujo"), hasta que alguien estornude junto al muro y éste se venga abajo.

El paralelismo con la programación es notable. Una palabra clave de BASIC es similar a la instrucción "construye un muro" que daríamos al albañil; esta orden lleva consigo el hacer una gran cantidad de trabajo, pero no necesariamente con la velocidad que desearíamos. Cuando no nos importe describir detalladamente cada acción, el códi-

go máquina será muchísimo más rápido, porque con él estaremos dando instrucciones a una máquina sin capacidad de pensar, pero increíblemente veloz: el microprocesador. Aún se puede llevar más lejos la similitud antes citada. Si usted dijese al albañil imaginario "repara el coche", éste podría no saber o no querer hacerlo; pero si proporciona al robot un conjunto de indicaciones específicas sobre cómo efectuar la reparación, el robot le aseguraría que la tarea se ejecutaría sin problema. El código máquina puede emplearse además para conseguir que el ordenador realice acciones que no permiten un rango de comandos mucho mayor que el que permitían los primeros modelos, y esta segunda aplicación del código máquina no es tan importante como lo era en otras épocas.

Antes de ocuparnos del modo en que el Spectrum funciona internamente, necesitamos mirar una vez más nuestro diagrama de bloques. El bloque con el rótulo "Puerto" corresponde a una gran cantidad de circuitos, que se encuentran contenidos en un único "chip" (circuito integrado). En terminología de los computadores, un PUERTO es algo que se dedica a transmitir información, de octeto en octeto, desde el interior del sistema de microcomputador al exterior, y viceversa (el sistema de microcomputador es el conjunto RAM, ROM y UCP). La razón de que estas entradas y salidas se manejen en una sección separada, es que son acciones muy importantes, pero *lentas*. Gracias a los puertos, podemos dejar al microprocesador elegir el momento en que quiere leer una entrada o escribir una salida. Por ejemplo, imagínese un programa en BASIC cuyas sentencias fuesen todas INPUT. Su velocidad de ejecución sería muy lenta, puesto que el programa se detendría y esperaría a que usted pulsase una tecla, seguida de ENTER, en cada línea. En cambio, si el programa contuviese únicamente una sentencia INPUT en la primera línea, podría ejecutarse sin interrupciones del teclado hasta el final del mismo. Puede comprobar que, si hace que el computador entre en un bucle sin fin con:

```
10 GOTO 10
```

entonces ninguna tecla sola tendrá efecto sobre la ejecución. A pesar de ello, el ordenador aún comprueba el puerto para saber qué hay en él, cada vez que ejecuta la instrucción, lo que puede deducir del hecho de que, al pulsar CAPS SHIFT y SPACE juntas, el bucle termina al interrumpirse el programa. La utilización del PUERTO es una forma de dejar al Spectrum seguir con el programa, permitiendo sólo este tipo de interrupción.

Además, no hay ninguna salida de resultados del ordenador, salvo cuando se obedecen las instrucciones PRINT o LPRINT, o cuando

finaliza un programa. Una vez más, no vemos nada nuevo en la pantalla mientras el microprocesador está trabajando. El puerto aisla la sección del computador encargada del manejo de la pantalla, manteniendo

```
1Ø FOR n = 16384 TO 22528
2Ø POKE n, RND*255
3Ø NEXT n
```

Fig. 1.9. Un programa que pone octetos de forma aleatoria en la pantalla.

Ya hemos visto las secciones más importantes que componen el corazón de su Spectrum. He empleado alguno términos con poca exactitud (por ejemplo, los formalistas pondrían objecciones al modo en que he usado la palabra "puerto"); pero no existe ninguna incoherencia en la descripción de cómo se ejecutan las acciones. Lo que tenemos que hacer ahora es ver de qué forma está organizado el computador para que, empleando la UCP, las memorias RAM y ROM, y el PUERTO, pueda ser programado en BASIC y ejecutar programas escritos en ese lenguaje. Este, pues, parece un buen sitio para empezar otro capítulo.

Capítulo 2

El Spectrum por dentro

No tome el título al pie de la letra: ¡no va a necesitar abrir la caja del ordenador! Lo que quiere decir es que en este capítulo vamos ver cómo está organizado el Spectrum para poder cargar y ejecutar programas en BASIC, y mientras lo hacemos, descubriremos el significado de algunos de los números y de los nombres crípticos que aparecen en el capítulo 25 del manual del computador.

Empecemos con una versión simplificada del funcionamiento de sistema completo (simplificada en el sentido de que omitiremos muchos detalles que sólo aportarían una gran confusión en esta etapa). La memoria ROM de su Spectrum consiste en un gran número de programas cortos (subrutinas), que están escritos en código máquina. Existe, por lo menos, una de tales subrutinas en código máquina para cada palabra clave de BASIC, y algunas de éstas necesitan utilizar muchas subrutinas. Cuando el Spectrum conecta la sección de programa en código máquina que se ejecuta, se llama "inicialización". Este es un trozo grande de programa, pero como el código máquina es rápido (se ejecuta a una velocidad de varios cientos de miles de instrucciones por segundo), usted podrá ver muy poco de lo que hace: la única prueba en la pantalla es el rectángulo negro que aparece justo antes de que aparezca el familiar mensaje de "copyright". A pesar de eso, en este breve lapsus de tiempo se ha comprobado el tamaño de memoria RAM existente (por si la noche pasada decidió añadir algún circuito más a ésta). Se han borrado todos los octetos indeseados, proceso que es necesario por los efectos que tiene el desconectar y volver a conectar la alimentación sobre la memoria. Cuando se desconecta la alimentación de una memoria RAM, todos los bits de ésta se ponen a Ø, como era de esperar. Al conectar de nuevo el suministro eléctrico, sin embargo, no existe ninguna garantía de que los bits permanezcan con ese valor de Ø. De hecho, aproximadamente la mitad de ellos se pondrán a "1" de un modo completamente aleatorio, de forma que si leyésemos cada octeto de memoria, en el instante en que acabamos de enchufar el computador, encontraríamos que cada uno de ellos tiene almacenado

un número entre Ø y 255, bastante al azar, y sin que se siga ninguna pauta concreta. Esta clase de cosas recibe el nombre de *información inválida* (en inglés se llama "garbage": ¡basura!), y una de las misiones del programa de inicialización es reemplazar todos esos octetos inválidos por un Ø, escribiendo dicho Ø en cada octeto de la RAM, uno tras otro. Como resultado de esa acción, si enchufa la máquina y comprueba (mediante PEEK) lo que hay en las direcciones de RAM superiores a 239ØØ, verá que el valor de cada octeto es cero.

Además de eso, la inicialización hace muchas cosas aún. De los 16k de RAM del Spectrum más pequeño, una gran parte es utilizada por el sistema operativo, lo que implica que las rutinas en código máquina utilizan la memoria RAM para guardar ciertos valores que pueden tener que cambiarse en varios momentos, mientras se ejecuta el programa de la memoria ROM. Las direcciones comprendidas entre 16384 y 23755 se emplean de ese modo, y eso son más de 7k de memoria (1k = 1024 octetos), que quedan fueran de los 16k, antes de haber pulsado un solo carácter de BASIC. Por añadidura, una vez que comienza a introducir un programa en BASIC, se gasta más memoria RAM, esta vez de las direcciones más altas de memoria, para almacenar una serie de valores necesarios en la ejecución posterior de su programa. Siempre que se declara una "variable", por ejemplo, con una línea así:

LET n = 2Ø o LET a$ = "Perez"

se toman varios octetos de memoria para guardar el nombre de la variable, n, a$ o cualquier otro nombre que use, y el valor de la misma, numérico o de cadena (en este caso 2Ø ó "Perez"). Con tales propósitos se utiliza una porción de memoria denominada Tabla de Variables (TDV), que se encuentra inmediatamente después del espacio ocupado por su programa. Si escribe una nueva línea de programa, la TDV se desplaza hacia direcciones superiores de RAM, y deja espacio libre para almacenarla. Cuando borra una línea, en cambio, la TDV completa se desplaza esta vez a posiciones más bajas de memoria, cubriendo el hueco que dejó la línea borrada.

Este tipo de funcionamiento debe controlarse cuidadosamente, ya que el computador tiene que saber en todo momento en qué dirección comienza la tabla de variables. Por esta razón existe un "índice" dentro de una zona de memoria reservada para el sistema operativo. Puesto que esta norma se utiliza bastante en el ordenador, podríamos examinar en detalle este ejemplo particular.

Las direcciones importantes que debe conocer el computador, se encuentran guardadas entre las posiciones 23552 y 23733, y la dirección

de comienzo de la tabla de variables, concretamente, está en las posiciones 23627 y 23628. Ya sabemos que un octeto sólo puede contener números de Ø a 255, y las direcciones como las anteriores tienen cinco cifras. Si se desea guardar números mayores que 255, al menos se precisa un octeto más, y la forma que tiene el computador para almacenar los números de dirección es utilizando dos octetos. El octeto extra va a contener el número de veces que la dirección contiene a 256, es decir, emplea una escala de 256, en vez de la de 1Ø ó la de dos. Si tuviésemos dos octetos almacenados con valores, por orden, 1Ø y 1, el significado de éstos sería 1Ø + 256 * 1 = 266. Si los octetos, por orden otra vez, fuesen 24 y 32, ambos a la vez representarían el número 24 + 256 * 32 = 8216. Observe con atención el orden de los octetos: primero es el octeto menos significativo y después el más significativo. Para descubrir el número de la dirección contenida en las posiciones 23627 + 256 * PEEK 23628. El resultado de esto es la dirección de comienzo de la tabla de variables.

Pruebe a introducir en el Spectrum:

```
1ØØ PRINT PEEK 23627 + 256 * PEEK 23628
```

y ejecútelo. Fíjese en el número que se obtiene. Ahora, añada algunas líneas tales como 1Ø LET n = 12 y 2Ø LET a$ = "Perez", y vuelva a ejecutar el programa. El número impreso será más grande, porque la dirección de comienzo de la tabla de variables se ha desplazado hacia arriba de la RAM, para dejar sitio libre a las dos líneas adicionales del programa BASIC. Si introduce más líneas de BASIC, el comienzo de la TDV se desplazará más arriba todavía.

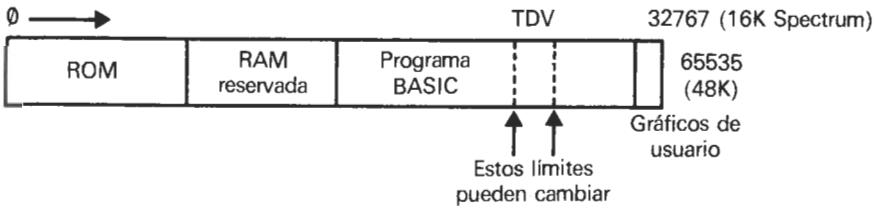


Fig. 2.1. Distribución de la memoria del Spectrum.

Si borra algunas líneas, entonces la TDV se desplazará otra vez hacia abajo de la memoria. Cuando las direcciones de la memoria que se asignan para algún propósito se almacenan de esta forma, decimos que hay "asignación dinámica de memoria". Debido a todo lo expuesto, el manual le previene que no se puede alterar la dirección de la

TDV (con un comando POKE), ya que esto haría que el computador perdiese la pista de alguna de las variables. Inténtelo: teclee el comando:

```
POKE 23627,2Ø3:POKE 23628,92
```

Ahora, ejecute el programa anterior y observe el caos que sobreviene. Desenchufe y vuelva a enchufar, si el Spectrum queda bloqueado y no responde al teclado. Lo que ha conseguido usted es confundir al veloz pero inconsciente microprocesador que hace funcionar a su Spectrum.

Esta es, en parte, una presentación del comando POKE. El primer número que sigue a POKE es la dirección cuyo contenido desea cambiar; el segundo número es el valor que quiere almacenar en esa dirección. Más adelante trataremos con más detalle el comando POKE; pero, con lo que ya sabe, fíjese que en el ejemplo de antes ponemos (usando POKE), el número 23755, principio de la zona de almacenamiento del programa BASIC, como dirección de comienzo de la tabla de variables.

Una vez que haya recuperado el control del ordenador, intente algo mucho más constructivo: vamos a examinar el contenido de la tabla de variables. Como usted habrá imaginado, tiene que contener el "nombre" de las variables y sus valores, pero el Spectrum codifica los valores de tal modo que pueda localizarlos y utilizarlos cuando sea necesario.

El primer número de la tabla de variables es siempre mayor que el 4Ø decimal. Esto se debe a que así el ordenador puede encontrar la *última* línea del programa BASIC. Los números de línea de los programas se restringen al rango de Ø a 9999 decimal, y como 9999 se codificaría en dos octetos, 15 (octeto menos significativo), y 39 (octeto más significativo), el octeto de mayor peso del número de línea nunca excederá a 39. Dado que el *primer* octeto de cualquier línea es el más significativo de los dos que componen el número de dicha línea, la máquina puede comprobar este octeto, y si es mayor o igual que 4Ø, querrá decir que ha encontrado el final del programa BASIC.

No obstante, eso significa que será necesario tener la precaución de que el primer octeto de la TDV sea un número mayor que 4Ø, lo cual es bastante sencillo, puesto que el código ASCII de un nombre de variable correcto debe ser un número más grande que 4Ø. Otro punto importante es codificar los nombres de las variables, de manera que el computador pueda distinguir los diferentes tipos de éstas. Esto se consigue utilizando sólo cinco bits del primer octeto para código de la letra del nombre de la variable, y los tres bits superiores para codificar

el tipo de la misma (numérica, de cadena, tabla, etc.). Vamos a explicar esto con más detenimiento.

En el caso de una variable numérica simple que posea un nombre de una sola letra, el primer octeto de su entrada en la TDV es precisamente el código ASCII del nombre. En el manual esto resulta algo confuso (lo que hacen es restar 96 del código ASCII, y después se lo suman de nuevo). Con ello se asegura que la entrada de la variable en TDV comienza con los bits Ø11, que es el código para variables numéricas con nombre de una sola letra. El valor de la variable está contenido en los cinco octetos siguientes al código de la letra. Salvo que usted sea curioso o tenga vocación de matemático, no precisa saber con demasiada exactitud cómo se almacenan números fraccionarios o negativos. Si verdaderamente le interesa, en el Apéndice B aparece desarrollado con todo detalle; pero si nos limitamos a números enteros, o sea, números positivos menores que 65535, en lo que se refiere al Spectrum, entonces cualquiera de ellos menor que 256 se almacena en un único octeto: el tercer octeto después del código de nombre. Si el entero está entre 256 y 65535, se utilizan dos octetos: el tercero contiene la parte menos significativa del número, y el cuarto, la parte más significativa (el número de veces que contiene a 256). En realidad, los enteros necesitan cinco octetos para almacenarse, y éste es uno de los factores que hacen al BASIC del Spectrum mucho más lento que el de las máquinas que trabajan de otra forma con enteros.

```
1Ø LET a=1Ø
2Ø LET b=11
3Ø LET c=12
4Ø FOR n=Ø TO 3Ø
5Ø PRINT PEEK ((PEEK 23627+256*PEEK 23628)
   +n) ;"        ";
6Ø NEXT n
```

Fig. 2.2. Programa para investigar el almacenamiento de los enteros.

La Figura 2.2 muestra un programa simple que le permite investigar el almacenamiento de algunos enteros y obtener en la pantalla los resultados. Si el nombre de la variable numérica consta de más de una letra, se hace indispensable otro tipo de codificación, puesto que, en caso contrario, el ordenador leería los cinco octetos que siguen al primer carácter como si fuesen el valor de la variable. En esta ocasión, lo que se hace es sumar 64 al valor ASCII de la primera letra del nombre, y después se guardan los caracteres siguientes en forma ASCII hasta el último, que tendrá 128 añadido a su código ASCII. Esto

permite al computador reconocer el final del nombre y tomar después los cinco octetos siguientes como el valor de la variable. La Figura 2.3 presenta un ejemplo de este proceso, que puede probar por sí mismo.

Es instructivo echar una mirada rápida a una variable que no sea entera, y la Figura 2.4 constituye un ejemplo. Ahora, lo interesante no es la codificación del nombre, sino el hecho de que los cinco octetos se emplean para almacenar el valor. Como consecuencia de esto (ver el Apéndice B para más detalles), el valor de la variable

```
10 LET julia = 23
20 FOR n = 0 TO 17
30 PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n) ;"        ";
40 NEXT n
```

Fig. 2.3. Forma de almacenar nombres de variable con más de una letra.

```
10 LET a = .5
20 LET b = .25
30 LET c = .125
40 FOR n = 0 TO 30
50 PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n)
60 NEXT n
```

Fig. 2.4. Variables no enteras. Se han elegido ejemplos que den resultados razonablemente simples.

que guarda el computador, nunca es exacto, y cuando su programa trabaje con números fraccionarios, existirán algunos "errores de redondeo". Esta circunstancia raras veces se percibirá en la pantalla, porque el valor que en ésta aparece no tiene tantas cifras decimales como el valor que está almacenado en la variable (cuando aparece en pantalla ya ha sido previamente redondeado), pero puede causar problemas en las sentencias BASIC que comparan valores para comprobar si son iguales.

Para hacerse una idea de lo que quiero decir, ejecute el programa de la Figura 2.5. En él, vemos que el computador no reconoce dos números como iguales, debido a una pequeña diferencia en los valores almacenados y, en cambio, aparecen escritos igual en la pantalla. Es una situación equivalente a decir que 1.00000007 es diferente a 1.00000008; pero si sólo escribimos los seis primeros decimales en la

pantalla, ambos números aparecerán como el 1.000000, es decir, la máquina detecta una diferencia de ¡una parte en un millón!

Nadie necesita tanta precisión, y los únicos problemas surgen en las pruebas de igualdad de los programas.

Una forma de tratar este problema es hacer las comparaciones después de haber redondeado siempre las cantidades que intervienen, con sentencias como:

```
IF INT (1Ø E4*a)=INT (1Ø E4*b) THEN GOTO...
```

que ejecutará el GOTO si los números son iguales en sus cuatro primeras cifras decimales (lo que suele ser suficiente para casi todos los propósitos).

```
1Ø LET n=1/1ØØØ
2Ø LET p=1Ø/1ØØØØ
3Ø PRINT "n=";n;" y p= ";p: IF n=p
   THEN PRINT "IGUALES"
4Ø IF n<>p THEN PRINT "DIFERENTES"
```

Fig. 2.5. ¡A veces, el ordenador no tiene el mismo concepto de igualdad que usted! Eso se debe a los «errores de redondeo».

Con el programa de la Figura 2.6, puede descubrir lo que ocurre cuando se almacenan enteros negativos en la TDV. No se preocupe de la forma exacta de codificación; únicamente observe que se utilizan tres octetos. Abandonando las variables numéricas, en la Figura 2.7, aparece otro nuevo programa para estudiar cómo se guardan las variables de cadena. El nombre de éstas se codifica con su valor ASCII menos 32, y el signo $ de cadena no se utiliza. Las variables de cadena del Spectrum sólo pueden tener nombre de una letra y, por tanto, en este caso no hay que codificar variables con nombres más largos. El

```
1Ø LET a=-12
2Ø LET b=-176
3Ø LET c=-255
4Ø FOR n=Ø TO 17
5Ø PRINT PEEK ((PEEK 2367+256*PEEK 23628)
   +n);"        ";
6Ø NEXT n
```

Fig. 2.6. Representación de los enteros negativos en la TDV.

```
1Ø LET a$ = "Sinclair"
2Ø FOR n = Ø TO 17
3Ø PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n);"        ";
4Ø NEXT n
```

Fig. 2.7. Almacenamiento de variables de cadena.

nombre de la variable va seguido de dos octetos que contienen el número de caracteres que posee, lo que indica a la máquina cuántos caracteres tiene que leer. Como la longitud de la cadena se conserva en dos octetos (la mayoría de los computadores sólo emplean uno para acelerar el manejo de cadenas), el Spectrum permite utilizar cadenas muy largas, a costa de desperdiciar un octeto cuando se están utilizando variables de este tipo, de longitud normal. Tras los octetos citados, vienen los caracteres que forman la cadena, almacenados en sus códigos ASCII corrientes. El Apéndice C explica el almacenamiento de las tablas numéricas y de caracteres. Esto es más complicado, y el tema se tratará de nuevo en el Capítulo 9 más detalladamente. Dentro de este apartado de almacenamiento de variables, podemos describir los números que habrá visto aparecer en la pantalla a continuación de las variables numéricas y de cadena que hemos estado estudiando. Precisamente, corresponden a los índices asociados a los bucles FOR... NEXT que hemos empleado para escribir los valores en cada caso. La Figura 2.8 es un programa que contiene exclusivamente uno de tales bucles, organizado de forma que imprima la parte de la TDV donde se encuentra almacenado su propio índice. El nombre de una variable índice de un bucle, en este ejemplo "n", está guardado como su código ASCII más 128, y su valor se conserva en los cinco octetos siguientes, del modo habitual. Dicho valor cambiará cada vez que se ejecuta el bucle y, debido a ello, el valor que verá en la pantalla será el que tenía la variable en el momento de imprimirlo. Como ese valor es el cuarto carácter de los que se imprimen (y n empezó valiendo Ø), lo que se obtiene resulta ser un 3, aunque n pasaría a valer 4 al ejecutarse la instrucción NEXT, y así sucesivamente. A continuación vienen cinco octetos más, reservados para el valor final del índice (18 en esta

```
1Ø FOR n = Ø TO 17
2Ø PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
   + n);"  ";
3Ø NEXT n
```

Fig. 2.8. Cómo se almacenan los índices de los bucles FOR.

ocasión), y un tercer grupo de cinco octetos, en los cuales está el valor asociado con STEP, que es 1, salvo que se especifique otra cosa. Cerca del final del bloque existen dos octetos para especificar el número de la línea a la que vuelve la ejecución, si no ha finalizado el bucle y el último octeto contiene el número de sentencia dentro de esa línea, en el caso de que el bucle FOR... NEXT haya comenzado dentro de una línea con varias sentencias del tipo de:

```
10 LET a=2 : LET b=3 : FOR n=1 TO 6
```

El número total de octetos ocupados es de 19; otra de las causas de la lenta velocidad de ejecución del BASIC del Spectrum. Muchas máquinas permiten sentencias como:

```
FOR N%=1 TO 16 STEP a%
```

que especifica dos enteros guardados en dos octetos cada uno; estos bucles se ejecutarán mucho más rápido y ocuparán mucha menos memoria. En el Spectrum no existe esta opción.

Almacenamiento de programas

Ya hemos investigado dos de las secciones de memoria RAM que controla el computador. El área más baja de la misma está reservado para uso del sistema, o sea: está reservado, tanto si escribe usted un programa BASIC, como si no lo hace. La parte superior de la RAM se utiliza principalmente con los programas BASIC, y cuanto menos necesite de ese espacio su programa, más memoria libre queda para el mismo. Es el momento de examinar más de cerca lo que sucede cuando se introduce un programa.

Como ya sabe, el computador presenta la línea que se está tecleando en la parte inferior de la pantalla, antes de pulsar ENTER. Se puede borrar parte de dicha línea, porque no se encuentra en su lugar de almacenamiento final, que empieza en 23755. En vez de eso, está guardada temporalmente en una porción de memoria denominada "tampón" (en inglés buffer). Esta es otra parte de la memoria asignada dinámicamente, que debe desplazarse hacia arriba, cada vez que se añade al programa BASIC una nueva línea. Debido en parte a esto, transcurre un tiempo apreciable desde que se pulsa ENTER, hasta que el cursor "K" reaparece en la línea inferior de la pantalla.

Suponga, por ejemplo, que tecleó:

```
10 LET n = 12
```

Esto se trata como una entrada temporal, hasta que usted pulse ENTER, y se encuentra colocado en la zona tampón. Si hubiese omitido el número de línea, ésta *nunca* habría pasado a otro lugar; pero con el número de línea presente, el computador está programado para colocarla en el espacio de memoria de programa cuando se pulse ENTER. ¿Dónde guarda el comienzo de esta memoria del programa? A pesar de que, generalmente, nunca varía, la dirección de comienzo de la zona de almacenamiento de programas se conserva en las posiciones 23635 y 23636, por lo que podemos encontrar la dirección del primer octeto del programa con la sentencia:

```
PRINT PEEK 23635 + 256*PEEK 23636
```

y el resultado es, normalmente, 23755. Si se cambia esta dirección escribiendo con POKE otros números en 23635 y 23636, su computador no reconocerá que ya hay un programa introducido, porque no puede darse un listado del mismo, a menos que la dirección donde aquel empieza esté presente. Es más, si la nueva dirección que se pone en esos dos octetos está en la RAM bastante más arriba de la que existía originalmente, se podría escribir un segundo programa en esta nueva dirección, listarlo y ejecutarlo. Cuando se restaurasen los octetos originales de 23635 y 23636, el primer programa podría ser listado y ejecutado, y trabajaríamos con él ignorando el segundo programa. Sin embargo, deberíamos asegurarnos de que cada programa tuviese su propia tabla de variables y que la dirección de ésta era la correcta, antes de ejecutar uno u otro.

Volviendo a la programación normal, ¿qué forma tiene un programa cuando se guarda en la zona de almacenamiento de programas? Se puede averiguar mediante un corto comando BASIC, suponiendo que no haya borrado aún la línea 10 anterior (es decir, 10 LET n = 12):

```
FOR n = 23755 TO 23785: PRINT PEEK n;" ";: NEXT n
```

Puesto que éste es un comando inmediato, no se mezclará con la línea que queremos investigar. La Figura 2.9 presenta lo que va a obtenerse con el comando citado: una serie de números, de los cuales la clave es el código ASCII de "n": 110. Mirando el listado del resultado, y sabiendo que 110 significa "n", podemos deducir que 241 es el código de LET (en el manual viene una lista de códigos de las

palabras clave), y ya sabemos que 61 es el código ASCII para el signo "=". Además podemos observar que el número 12 se almacena como dos octetos en código ASCII: 49 para el "1", y 5Ø para el "2". Con eso hemos descubierto ya que LET n = 12 ocupa cinco octetos; pero aparecen cuatro más delante de LET y un gran número de ellos detrás, con un 13 al final de la cadena de números.

```
Ø   1Ø   12   Ø   241   11Ø   61   49   5Ø
14  Ø    Ø    12  Ø     Ø    13   234
Ø   Ø    222  92  Ø     Ø    Ø    233  92
Ø   Ø    Ø    1   Ø
```

Fig. 2.9. Aspecto de una línea BASIC en la memoria.

El octeto final es 13, el código que se usa para la tecla ENTER y los octetos que hay entre 14 y 13 constituyen la forma codificada empleada en la tabla de variables: 14 es la señal de que lo que sigue está en forma correcta de código numérico, que es diferente del código ASCII. Los cuatro octetos que preceden al código de LET nos interesan más en este momento, porque ilustran una nueva forma que tiene el ordenador de utilizar la RAM para saber lo que pasa en cada momento. Los dos primeros octetos son el número de línea. A diferencia de los demás números de dos octetos que maneja el ordenador, éstos están en orden normal (para nosotros): la parte más significativa primero (las veces que el número contiene a 256) y después la parte menos significativa. La razón de hacerlo así, es que el computador pueda encontrar el fin de programa cuando lea un valor mayor que 4Ø (como ya explicamos antes). En este caso particular no tendría importancia, pues los octetos son 1Ø y Ø; pero si hubiese comenzado con un número de línea 1ØØØ, los dos octetos habrían sido 232 y 3, porque 3*256 + 232 es igual a 1ØØØ.

¿Para qué son los dos números siguientes? Si introdujo usted la línea tal como yo la puse, los dos octetos siguientes serán 12 y Ø. Esos dos se encuentran de nuevo en la forma usual, es decir, parte menos significativa seguida de la más significativa, y el 12 corresponde al número total de octetos de código que componen la línea, incluyendo el texto de la misma (LET n = 12), la entrada de TDV para "n" y los cinco octetos con su valor y el código 13 de fin de línea, pero sin contar los cuatro octetos del comienzo. ¿Por qué necesita conocer esta longitud? Es muy simple, como todas las acciones del microprocesador. Si el computador lleva la cuenta de los octetos que va leyendo, entonces el número contenido en su contador de direcciones, una vez que haya leído estos dos octetos de "longitud de línea", será la dirección del

primer octeto útil de la línea de programa (que en este caso correspondería a LET). Cuando sume la longitud de dicha línea al contador de direcciones, en éste quedará la dirección de comienzo de la línea siguiente de programa, que empieza (como la anterior) con su número y longitud. Es así como el ordenador pasa de una línea a la siguiente, transfiere el número correcto de octetos a la parte inferior de la pantalla (que es otra zona de la memoria) para edición de líneas, y puede reordenar éstas de manera que queden almacenadas en secuencia ascendente de sus números.

Cada vez que usted teclea una nueva línea de BASIC y la introduce pulsando ENTER, está utilizando cinco octetos de memoria para el número y la longitud de la misma y el octeto con el código de ENTER (que es 13). A esa parte se la denomina "cabecera de línea", y lo interesante aquí es el uso de dos octetos para la longitud de línea. La mayor parte de los computadores emplean sólo un octeto para ese cometido, lo que no permite líneas con más de 255 caracteres, que es lo más normal y, por lo tanto, en el Spectrum, uno de los dos octetos de longitud será cero casi siempre. Se desperdicia un octeto por cada línea, a menos que tenga usted la costumbre de escribir líneas enormemente largas. Puede tratar de evitar algo de este desperdicio de octetos en las cabeceras, utilizando, con frecuencia, líneas con varias sentencias, porque los octetos de número y longitud de las líneas se colocan cada vez que comienzan éstas, no cuando se separan sentencias dentro de ellas con los dos puntos. Pruebe lo siguiente:

```
1Ø LET a=12: LET g$="Sinclair"
```

y después examine la línea por medio del comando:

```
FOR n=23755 TO 23785: PRINT PEEK n;" ";NEXT n
```

que le descubrirá cómo se almacenó aquella. El resultado aparece en la Figura 2.10. Ahora, escriba POKE 23756,2Ø (y ENTER, claro) y liste el programa.

```
Ø    1Ø   27   Ø    241  97   61   49   5Ø
14   Ø    Ø    12   Ø    Ø    58   241
97   36   61   34   83   1Ø5  11Ø  99
1Ø8  97   1Ø5  114  34   13
```

Fig. 2.10. El resultado de almacenar una línea con dos sentencias.

Ha cambiado el número de la línea 1Ø a 2Ø, alterando únicamente el octeto de número de la misma. Introduzca algún otro comando POKE. Por ejemplo:

```
POKE 23759,234 : POKE 23760,13 : POKE 23752,2 :
POKE 23761,238
```

y luego pulse ENTER y liste el programa. Este aparecerá ahora así:

```
2Ø REM
```

No queda ningún rastro del programa original, aunque sus octetos están todavía en la memoria. Esto ilustra de un modo espectacular lo que POKE, que coloca octetos en la memoria directamente, puede hacer. Posteriormente, veremos la sintaxis de este comando.

Ejecución de un programa

Una vez que haya introducido las líneas de un programa BASIC en la memoria, éste se encontrará almacenado como acabamos de ver, con los octetos de número y longitud de línea, los tokens de las palabras claves y códigos ASCII que componen las líneas, los códigos de entrada a la tabla de símbolos y el número 13 que marca el fin de cada línea. Cuando se introduce cada línea, lo cual incluye el transferirla desde la zona tampón a su lugar definitivo de almacenamiento, la sintaxis de ésta es comprobada y, si se detecta algún error, se enviará el mensaje habitual de aviso de sentencia incorrecta. La familia de computadores ZX es casi la única que lleva a cabo la comprobación sintáctica antes de que se introduzca una línea, y esto puede ahorrar un trabajo posterior bastante aburrido. Muchos ordenadores aceptan alegremente errores de sintaxis, y sólo informan de los mismos cuando se intenta ejecutar el programa, lo que resulta un poco tarde. Ahora, no obstante, queremos estudiar la forma en que el computador maneja las líneas BASIC que están codificadas y almacenadas en memoria, a partir del momento en que se introduce RUN seguido de ENTER. Antes de que usted pase a esta etapa, recuerde que la máquina ha guardado una gran cantidad de información relacionada con su programa, en secciones reservadas de su memoria RAM. Por ejemplo, todas las variables estarán listas para ser colocadas en la TDV, cuya dirección de comienzo se ha calculado y almacenado previamente. La posición de la primera línea BASIC está contenida en las direcciones 23635 y 23636, y el final de los octetos del programa se encuentra donde empieza la TDV, como vimos.

Los computadores siguen unas reglas inflexibles. El primer octeto de una línea, después de los octetos de número y longitud de ésta que forman parte del sistema auxiliar de la máquina, es siempre un token de palabra clave. El Spectrum asegura esta circunstancia mediante su sistema de entrada de líneas, que comprueba cada una y rechaza las que son incorrectas. De esta manera, se evita tener que comprobar cada línea durante la ejecución del programa (como hacen otros ordenadores). Lo que sucede después, depende de la instrucción en sí. Si es una asignación simple, como LET n = 12, entonces se copia en la tabla de variables la parte de la línea que comienza con el número de código 14.

Una asignación compleja, como LET y = 2*n, precisa una atención mayor. El token de LET provoca una llamada a una subrutina en código máquina que introduce "y" en la tabla de variables; pero esta rutina debe interrumpirse para ejecutar otra de multiplicación y obtener el resultado de 2*n. Esta última emplea, a su vez, un conjunto de subrutinas en código máquina, que buscan primero el valor de n a través de la TDV, extraen dicho valor, realizan la multiplicación y devuelven la ejecución a la primera de todas, que se encarga de almacenar el resultado en el lugar correspondiente creado para la variable "y", codificado en los cinco octetos que siguen al nombre. Los dos ejemplos anteriores presentan dos acciones importantes que tienen lugar durante la ejecución (acciones en tiempo de ejecución): almacenar valores en la tabla de variables y leer en ella otros valores almacenados previamente. Una gran parte de las acciones simples de BASIC son de uno de esos tipos. Por ejemplo, el comando PRINT a$ comenzará con el token de PRINT que causa una llamada a una rutina de impresión (una de las más extensas y complejas de la ROM). A su vez, ésta llamará a otra, que se ocupa de buscar el valor de a$ en la TDV y, una vez encontrado, almacenará el número de caracteres que posee junto con la dirección donde empieza su primer carácter. Después de eso, la máquina sabe la longitud de la cadena y dónde está almacenada, con lo que puede completar la rutina de impresión. Esta es bastante compleja, debido al diseño de los computadores modernos como el Spectrum. En otros tiempos, la rutina "PRINT" solamente tenía que copiar los octetos de los caracteres desde la memoria de programa o la TDV, a un conjunto de direcciones de memoria llamado memoria de vídeo. Esta era una zona de memoria separada físicamente del resto, de modo que un computador como el TRS-8Ø de 16k presentaba realmente 16k de RAM, libres para el usuario (la memoria para la pantalla, que era de 1k, estaba separada y no contaba en esa cantidad total). La técnica que impera en la actualidad, es utilizar una gran cantidad de RAM para lo que el manual del Spectrum denomina un "fichero de

pantalla" (otra variante de memoria de vídeo). Este forma parte de los 16k de memoria RAM, y su misión es almacenar los octetos que describen la forma de los caracteres y, en el Spectrum, cada carácter necesita ocho octetos de RAM. Los octetos citados, sin embargo, no se guardan consecutivos en la memoria. Los ocho octetos para cada carácter se encuentran, cada uno, 256 lugares más allá del anterior. Puede comprobar esto por sí mismo con el programa BASIC de la Figura 2.11, cuyo trabajo consiste en poner (con POKE) octetos en las localidades de memoria que corresponden a la primera posición de carácter de la pantalla.

Los octetos que componen los caracteres impresos en el teclado están contenidos en la memoria ROM y, como era de esperar, la dirección de comienzo de este conjunto de octetos, el juego de caracteres, se conserva en la RAM reservada para el sistema operativo. La dirección almacenada en 236Ø6 y 236Ø7 es el principio del juego de caracteres ya mencionado, menos 256 (por ello, el programa que emplee esa dirección puede ejecutar un bucle que empiece sumando 256 a la misma), que da como resultado la posición 15616. El juego de caracteres se encuentra en la ROM; pero, ya que su dirección de comienzo está contenida en RAM, podríamos cambiar esta última y hacer que apuntase a un nuevo conjunto de caracteres creado por nosotros y que hubiésemos almacenado en la memoria previamente.

```
1Ø FOR n=Ø TO 7
2Ø POKE 16384+256*n, 68
3Ø NEXT n
```

Fig. 2.11. Escritura de octetos en el fichero de pantalla. Es necesario poner los valores que queremos escribir, en posiciones de memoria separadas 256 octetos.

Precisamente, eso es lo que hacemos cuando creamos "gráficos definidos por el usuario". La rutina de impresión hace uso de esos octetos de carácter almacenados, y envía copias de ellos a las direcciones apropiadas del fichero de pantalla, que se encuentra al principio de la memoria RAM, con lo cual los circuitos dedicados a la televisión pueden generar las señales que dan forma a los caracteres en su receptor. Al mismo tiempo, las rutinas "auxiliares" entran en acción para asegurar que el siguiente carácter aparecerá en la posición actual de impresión, o en cualquier otra que especifiquen las funciones TAB o AT, o al principio de una nueva línea. Sería posible escribir volúmenes enteros para cada acción del Spectrum examinada detalladamente; pero no tiene interés para nosotros, porque todas las acciones siguen una pauta bastante general. Una palabra de comando se representa en

la memoria con un token que se utiliza en tiempo de ejecución para llamar una subrutina que emplea por turno la tabla de variables, y otras subrutinas, para completar su trabajo. Algunas veces, una de esas rutinas debe "esperar" (por ejemplo, una subrutina para imprimir n*k no podrá terminar su labor hasta que se calcule primero el valor n*k). El computador tiene que prever esa circunstancia, y, para ello, usa la memoria RAM como almacenamiento temporal, de manera que existe cierto número de octetos reservados para ese propósito.

Finalmente, ¿cómo sabe el ordenador a qué rutina hay que llamar cada vez? Es bastante sencillo, cuando se sabe. Los tokens de las palabras reservadas, se encuentran en la memoria ROM almacenados por orden. Para buscar uno de ellos, se empieza por el primero y se van comprobando uno tras otro, llevando un contador. Cuando se encuentra el token buscado, el computador tendrá un octeto con la cuenta (en la parte de RAM reservada) de a cuántas posiciones más allá del principio de la tabla se encuentra situado el token. Este número del contador se suma a otro número de dirección (contenido en la ROM), y el resultado es el lugar de almacenamiento de la dirección de la rutina correcta. En la práctica, no es tan simple, pero el principio es el mismo: encontrar un dato en una lista proporciona un número de cuenta, que, posteriormente, sirve para descubrir una dirección en otra lista.

Capítulo 3

La unidad central de proceso

En este capítulo, estudiaremos con detenimiento el microprocesador Z-8ØA del Spectrum. El microprocesador o UCP (unidad central de proceso) es, recuerde, la parte "activa" del computador, a diferencia de la parte de almacenamiento (memoria) o la de entrada/salida (el PUERTO), de tal manera que el microprocesador decidirá lo que hace el computador en todo momento.

La UCP es un conjunto de posiciones de memoria, pero con una gran cantidad de elementos adicionales. Haciendo uso de unos circuitos denominados (con gran acierto) puertas, se puede cambiar y controlar la forma en que los octetos se transfieren de un sitio a otro de la memoria interna de la UCP. Estas acciones son las que constituyen la suma, la resta, las operaciones lógicas y otras operaciones adicionales del microprocesador. Cada acción debe ser programada, y no ocurrirá nada, a menos que esté presente un octeto de instrucción (en forma de señales "Ø" ó "1" en cada uno de los ocho terminales adecuados de la UCP). Con esos octetos se controlan las puertas que existen en el interior de la UCP y que ya hemos mencionado. Lo que hace que el sistema sea tan útil es que las instrucciones de programa están en forma de señales eléctricas en ocho líneas, y por eso pueden cambiarse muy rápidamente. Esta velocidad depende de otro circuito electrónico llamado "generador de pulsos de reloj" (o simplemente "reloj", para abreviar). El Z-8ØA puede trabajar con un reloj de 4MHzs., lo que significa que genera cuatro millones de pulsos por segundo. El microprocesador, pues, ejecutará sus operaciones internas a esa velocidad; pero debido a que una instrucción completa puede requerir varias operaciones internas, la velocidad de ejecución de instrucciones en código máquina completas es menor que la velocidad del reloj (un valor típico es una velocidad de medio millón de instrucciones por segundo). El Spectrum lleva un reloj que funciona a 3.5MHzs, es decir, produce tres millones y medio de pulsos por segundo.

Circuitos y programas

A las partes eléctricas (circuitos) del computador se las denomina material* (hardware en inglés). Cambiar el diseño eléctrico puede resultar bastante difícil: cortar cables aquí y soldar otros por allá, dentro de una maraña de conexiones que incluso el propio diseñador tendría dificultades para reconocer. El programa que consigue que los circuitos actúen como un ordenador es un ejemplo de **logical*** (software en inglés). Este último es infinitamente más fácil de alterar, ya que en el caso de Spectrum, está completamente contenido en un circuito integrado de memoria ROM; si cambia ese circuito ¡tendrá un computador totalmente distinto! Algunos ordenadores "familiares", poseen una memoria ROM muy pequeña; prácticamente todo su logical se lee de cintas o discos, y puede cambiarse con suma facilidad. Con este método, si usted se cansa de BASIC, puede cambiar de lenguaje inmediatamente, cargándolo en la memoria RAM. A pesar de ello, la mayoría de nosotros preferimos el sistema de BASIC en circuito ROM, que está asegurado contra los desastrosos efectos de los comandos POKE erróneos.

En lo que se refiere a la UCP, el logical es una colección de octetos, a la que damos el nombre de código máquina. Un programa escrito en código máquina es, para nosotros, una lista de números, cada uno de los cuales tiene un valor entre 0 y 255. Algunos de esos números pueden ser octetos de instrucción, que ordenan a la UCP que haga algo. Otros pueden ser octetos de datos, o sea, números para sumar o almacenar, o tal vez códigos ASCII. El microprocesador no distingue las instrucciones de los datos, y es misión del programador asegurarse de que todo se haga correctamente, poniendo los números en el orden adecuado.

El orden correcto, desde el punto de vista de la UCP, es muy elemental. El primer octeto que se proporcione a ésta, después de conectar el computador o después de terminar la instrucción anterior, es tratado siempre como una nueva instrucción. Algunas de esas instrucciones consisten en un solo octeto, y otras necesitan a continuación dos o más octetos, que serán de instrucción o de datos. Si tomamos las funciones RND y TAB de BASIC, recordará que RND se puede utilizar sola (aunque puede multiplicarse por otro número), pero TAB debe ir precedida por el comando PRINT y seguida de un número. Cuando la UCP recibe un octeto de instrucción, obtiene de él informa-

* Vamos a emplear **material** y **logical** como traducción de las palabras **hardware** y **software**, respectivamente, por ser los términos que parecen imponerse, por fin, en castellano, para designar esos dos importantes conceptos de informática. (N. de T.)

ción sobre los octetos que vienen a continuación. Por lo tanto, los octetos de instrucción pueden ser de cuatro tipos:

a) Los que especifican una operación por sí mismos, b) aquellos que necesitan un octeto de datos adicional, c) los que necesitan dos octetos de datos adicionales y d) algunos que necesitan un octeto extra de instrucción, que, a su vez, puede ir seguido por octetos de datos. Cada uno de los octetos de instrucción lleva, intrínsecamente, información codificada que permite al microprocesador determinar lo que viene a continuación.

El inconveniente es que el programador debe hacer todo correctamente: ¡como mínimo debe hacer las cosas con un 100% de exactitud! Si se proporciona al microprocesador un octeto de instrucción cuando está esperando un dato, o un octeto de datos cuando necesita una instrucción, los resultados pueden ser desastrosos. Los desastres aludidos pueden consistir en un bucle sin fin, que provoque la pérdida de imagen en la pantalla del televisor y el bloqueo total del teclado (incluso de la tecla BREAK), y, en consecuencia, habrá que desconectar el ordenador y conectarlo de nuevo. Otra posibilidad es que la máquina entre espontáneamente en la rutina de inicialización, borrando toda la memoria y presentando el mensaje de "copyright". En muchos de los casos, se perderán los programas que existiesen en memoria (la pérdida es segura cuando hay que desenchufar para recuperar el control), y la "moraleja" es que usted debería grabar el programa en cinta de cassette o en microdrive, ¡antes de ejecutarlo!

En este punto quiero hacer hincapié en que la programación en código máquina es aburrida. No es difícil (consiste en formalizar un conjunto de instrucciones simples para una máquina simple), pero, a menudo, no resulta fácil recordar todos los detalles necesarios. Cuando programa en BASIC, los mensajes de error del computador le ayudarán a descubrir y corregir las equivocaciones; por el contrario, cuando escriba código máquina, todo correrá de su cuenta y deberá encontrar su propios errores, aunque el empleo de un ensamblador ayuda considerablemente. Como la mejor forma de conectar el código máquina es escribir y usar dicho código, y aprender de sus inevitables errores, dedicaré el resto de este capítulo a los métodos de escribir números en código máquina, incluso antes de que aprenda lo que significan y qué instrucciones están disponibles, o cómo se usan. Comenzaremos con las formas que existen para escribir los números que constituyen los octetos de los programas en código máquina.

Números binarios, decimales y hexadecimales

Un programa en código máquina consiste en un conjunto de códigos numéricos. Puesto que cada número es una manera de representar los unos y ceros de un octeto de memoria, consistirá en números dentro del rango de Ø a 255, cuando escribimos en base diez (base decimal). El programa no sirve para nada hasta que puede almacenarse en la memoria del Spectrum, porque el microprocesador es un dispositivo rápido, y la única forma de proporcionarle octetos a la velocidad a la que los necesita, es almacenarlos en una memoria RAM o ROM, y dejar al microprocesador que los vaya leyendo en orden cada vez que sea preciso. Posiblemente, no podrá teclear números con rapidez suficiente para satisfacer a la máquina, e incluso los medios de almacenamiento como cintas y discos tampoco son suficientemente rápidos.

En consecuencia, introducir octetos en la memoria es una parte esencial del proceso de creación de programas útiles en código máquina, y, en el Capítulo 5, veremos con más calma los métodos para lograrlo. En tiempos pasados, los programas cortos y simples se escribían, en la memoria de los sistemas sencillos de microprocesador, del modo más primitivo posible: se disponía de ocho interruptores, cada uno de los cuales podía colocarse para dar una salida 1 ó Ø, y de un botón que provocaba la lectura en memoria del número binario que representaba el estado de los interruptores. Además, se hacía necesario tener algún medio de direccionar la memoria, y eso podía realizarlo el propio microprocesador, como veremos después.

Programar con ese procedimiento es demasiado tedioso, y trabajar directamente con números binarios conduce a equivocaciones sin fin. Teniendo en cuenta que un número binario es un conjunto de ceros y unos, después de leer e introducir unas cuantas docenas de ellos, se empieza a cometer errores, cambiando los ceros y unos, repitiendo números, etc... El paso evidente es aprovechar el computador para colocar los números en su memoria, y otra mejora obvia es utilizar una base de numeración más conveniente.

La elección de una base de numeración es un asunto que depende de cómo introduzca usted los números y con qué frecuencia programe en código máquina. Un computador como el Spectrum contiene subrutinas que convierten números binarios a una forma que le permite imprimir automáticamente números decimales en la pantalla, y pueden llevar a cabo, también, la transformación inversa. Cuando emplea PEEK, entonces la dirección que le sigue está en decimal, y el resultado de esa función será un número decimal entre Ø y 255. Si emplea POKE, puede escribir ambos números, dirección y valor, en decimal. Por tanto, tiene sentido trabajar con el sistema de números al que

estamos acostumbrados y que el propio Spectrum utiliza para sus comandos PRINT e INPUT.

Sin embargo, los buenos programadores de código máquina encuentran este sistema demasiado inflexible. Hasta la fecha, el mejor procedimiento para introducir programas en código máquina es escribirlos en lo que se llama **lenguaje ensamblador,** cuyos comandos son fragmentos de palabras. Se pueden escribir programas denominados **ensambladores,** que convertirán esos comandos en los códigos binarios correctos. De este modo, los programadores nunca tienen que preocuparse por los códigos binarios reales: la mayoría de los ensambladores presentarán los códigos en la pantalla escritos en base hexadecimal. Esos ensambladores requerirán que los números que se les proporcione sean también hexadecimales.

Código hexadecimal

Hexadecimal significa base de numeración (o escala) dieciséis, y la razón de que se emplee tan exhaustivamente es que resulta apropiado, por su naturaleza, para representar octetos binarios. Cuatro bits (dígitos binarios), o sea, la mitad de un octeto, representarán números comprendidos entre Ø y 15 en nuestra base habitual, y ése es, precisamente, el rango que abarca un dígito hexadecimal (ver Figura 3.1). Esto significa que un octeto puede representarse con dos de tales dígitos; y una dirección de dos octetos, con cuatro dígitos hexadecimales (a veces emplearemos hexa para abreviar, a partir de ahora). Adicionalmente, es mucho más fácil relacionar un número hexa con la tira de bits que componen un octeto, de lo que resultaría si se trabajase con base decimal. Además, los códigos numéricos que constituyen los octetos de instrucción de un microprocesador se escriben generalmente con dígitos hexa, y éstos permiten ver más claramente la secuencia de organización de instrucciones, cuando, por ejemplo, un conjunto de órdenes relacionadas empiezan todas con un mismo dígito hexa.

En el momento de escribir este libro, acababa de aparecer un ensamblador para el Spectrum, que admitía números decimales, aunque presentaba los resultados en hexadecimal. Los desensambladores tales como los de Campbell Software (DPAS) y ACS Software (INFRARED), muestran los códigos numéricos y los datos en hexadecimal. A medida que vaya progresando en el dominio del código máquina, se hará evidente la necesidad de conocer el sistema hexadecimal de numeración, incluso si usted evita su utilización. Puede escribir sus programas en código máquina para el Spectrum en base decimal; pero, si cambia de computador, comprobará que es indispensable manejar

Hexa	Decimal
Ø	Ø
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	1Ø
B	11
C	12
D	13
E	14
F	15
Luego	
1Ø	16
11	17
Hasta	
2Ø	32
21	33
etc.	

Fig. 3.1. Números hexadecimales y decimales.

código hexadecimal. Por otro lado, hay que decir que los libros más avanzados sobre programación en código máquina, algunos de los cuales se citan en el Apéndice A, suponen que usted está habituado al manejo de números hexadecimales.

La base hexadecimal: conversiones

La base hexadecimal consta de dieciséis dígitos, que comienzan con el Ø y siguen en orden ascendente hasta el 9, como en los números decimales. El siguiente dígito, no obstante, no es 1Ø, puesto que en esta base eso representa al dieciséis, y debido a que no poseemos símbolos para dígitos por encima del nueve, tenemos que hacer uso de las letras de A á F. El número decimal 1Ø se escribe ØA en hexa, once es ØB,

doce es ØC, y así sucesivamente hasta quince, que es ØF. El cero no necesita escribirse, pero los programadores tienen la costumbre de escribir octetos de datos con dos dígitos, y las direcciones con cuatro, aun cuando algunas veces pueda hacerse con menos dígitos. El número siguiente a ØF es 1Ø, dieciséis, y después la secuencia es similar hasta llegar a 1F, treinta y uno, que va seguido por el 2Ø. El máximo valor de un octeto, 255, es en hexa FF. Cuando se emplea esta base, es costumbre escribir una "H" después del número, para no confundirlo con un número decimal. Un número tal como 16 podría ser dieciséis (decimal) o veintidós (hexa), pero no hay duda posible si se pone 16H.

El gran valor de esta base es la facilidad de pasar de ella a código binario y viceversa. Si observa la tabla binaria-hexadecimal de la Figura 3.2, puede comprobar que 9 es 1ØØ1 en binario, y F es 1111. El número hexa 9FH es precisamente 1ØØ11111 binario: simplemente se escriben los cuatro bits a que equivale cada dígito hexa, debajo de éste. La conversión en dirección opuesta es igual de sencilla: se agrupan los dígitos binarios de cuatro en cuatro, empezando por el menos significativo (o sea el bit más a la derecha), y después se convierte cada grupo de cuatro en el dígito hexadecimal correspondiente. La Figura 3.3 muestra ejemplos de conversión en ambas direcciones, y puede ver en ella lo simple que resulta.

El manejo de números hexadecimales varía mucho de un computador a otro. Algunos como el Spectrum, no pueden tratar esos números directamente, y si alguien quiere llevar a cabo la programación en

Hexa	Binarios
Ø	ØØØØ
1	ØØØ1
2	ØØ1Ø
3	ØØ11
4	Ø1ØØ
5	Ø1Ø1
6	Ø11Ø
7	Ø111
8	1ØØØ
9	1ØØ1
A	1Ø1Ø
B	1Ø11
C	11ØØ
D	11Ø1
E	111Ø
F	1111

Fig. 3.2. Números hexadecimales y binarios.

Conversión: **hexadecimal-binario**

Ejemplo: 2CH ..... 2H es ØØ1Ø binario
CH es 11ØØ binario

Luego 2CH será ØØ1Ø11ØØ binario

Ejemplo: 4A7FH ..... 4H es Ø1ØØ binario
AH es 1Ø1Ø binario
7H es Ø111 binario
FH es 1111 binario

Luego 4A7FH será Ø1ØØ1Ø1ØØ1111111 binario

Conversión: binario-hexadecimal

Ejemplo: Ø11Ø1Ø11 ..... Ø11Ø es 6H
1Ø11 es BH

Luego Ø11Ø1Ø11 será 6BH

Ejemplo: 1Ø11Ø11ØØ1ØØ1Ø ..... vea que no hay un número exacto de octetos.

Se agrupan de cuatro en cuatro, empezando por el menos significativo (bit más a la derecha):

ØØ1Ø es 2H
1ØØ1 es 9H
11Ø1 es DH y
el resto, 1Ø, es 2H

Luego, el número completo es 2D92H

Fig. 3.3. Cambios de base decimal y binaria.

código máquina en base hexadecimal, tendrá que utilizar un ensamblador que acepte números en esa base. En cambio, computadores como el BBC, tienen un traductor hexadecimal incorporado; la máquina BBC posee incluso un ensamblador en su memoria ROM.

Supongamos, de momento, que usted no dispone de ensamblador para crear los programas en código máquina: ¿qué puede hacer? La respuesta es que diseñará su programa en lenguaje ensamblador, que

es, por ahora, la forma más fácil de crear programas en código máquina, y después lo convertirá a código decimal en vez de teclear las instrucciones en lenguaje ensamblador. Esa conversión significará localizar (en una serie de tablas llamadas **juego de instrucciones**) el número que representa cada instrucción.

Los juegos de instrucciones que proporcionan los fabricantes de microprocesadores o los fabricantes de computadores, están casi siempre en hexadecimal; pero el manual del Spectrum da una lista de los códigos también en decimal, aunque en orden numérico, en vez del orden alfabético, que sería el más apropiado para nuestros fines. Para ayudarle, se ha incluido una lista completa de códigos del Z-80 ordenados alfabéticamente, en el Apéndice F de este libro, empleando las bases decimal, hexadecimal y binaria. ¡No la examine en este momento, porque le desanimará!

La mayoría de los programas en código máquina poseen octetos de datos además de instrucciones, y aquellos también se presentan a menudo en hexadecimal, aunque es igual de fácil presentarlos en decimal. A causa de esto, es útil saber pasar de base decimal a hexadecimal, y viceversa. Si usted posee el Desensamblador DPAS de Campbell Software, podrá utilizar el programa de conversión decimal-hexadecimal que viene incluido. En caso contrario, tendrá que hacer la conversión "a mano" o ejecutar un programa simple que lo haga, cada vez que trabaje con esos códigos.

El paso de base hexadecimal a base decimal aparece en la Figura 3.4. Para números de un octeto el método es muy simple: se toma el valor decimal del dígito hexa más significativo, se multiplica por 16, y se le suma el valor del otro dígito hexa. Los números de dos octetos, como las direcciones, son más aburridos. Si el número hexa tiene cuatro dígitos, el valor del más significativo se multiplica por 16 × 16 × 16, cuyo resultado es 4096 (todo en decimal). Se anota este valor, y el valor del dígito siguiente se multiplica por 256 y se anota también. El dígito hexa que viene después se multiplica por 16 y se anota el resultado, y el dígito menos significativo se escribe debajo. Finalmente, sumamos todos los resultados parciales que hemos anotado y obtenemos el equivalente decimal del número original completo. Se muestran ejemplos en la Figura 3.4.

El paso de decimal a hexa, ya no es tan simple. Un número de un sólo octeto se divide por 16, lo que da una parte entera y una fraccionaria. La parte entera se convierte a un dígito hexadecimal, y la parte fraccionaria se multiplica por 16, y el resultado se transforma en otro dígito hexa. Para direcciones (con 4 octetos), la división del número decimal dado por 16 dará una parte entera que es mayor que 16. Se convierte la parte fraccionaria en un dígito hexa, multiplicándola por

CONVERSION HEXADECIMAL-DECIMAL

(a) Un solo octeto.

Ejemplo: convertir 3DH a decimal. El valor es 3*16+13 (D en decimal) que da 61 decimal.

Ejemplo: convertir A8H a decimal. El valor es 1Ø*16+8, que da 168 decimal.

(b) Dos octetos.

Ejemplo: convertir 2CA5 a decimal. El primer dígito da 2*4Ø96=8192. El segundo dígito da 12*256=3Ø72, y el tercer dígito da 16*1Ø=16Ø. El dígito final es 5, y sumando 8192+3Ø72+16Ø+5 obtenemos 11429.

Ejemplo: convertir F3DBH a decimal.

	1.er dígito	15*4Ø96	= 61440
	2.º dígito	3*256	= 768
	3.er dígito	13*16	= 2Ø8
	4.º dígito	11	= 11
v02n	Suma	.....	= 62427

Fig. 3.4. Conversión hexadecimal-decimal para octetos simples o dobles.

16, y después se repite el mismo proceso con la parte entera que resultó de la división, o sea, se divide por 16, escribiendo debajo la parte entera, y pasando la nueva parte fraccionaria a dígito hexa, multiplicándola antes por 16. Este procedimiento se sigue repitiendo hasta que, en una división, la parte entera sea menor que 16, y entonces, se escribe su dígito hexa equivalente y termina el método. La Figura 3.5 presenta algunos ejemplos de este cambio de bases.

Es más fácil utilizar programas de conversión. El paso de decimal a hexa se puede hacer de modo simple, dividiendo sucesivamente por 4Ø96, 256 y 16, y convirtiendo la parte entera del resultado a dígito hexa cada vez. La conversión se realiza empleando códigos ASCII, porque cuando su Spectrum imprime un número hexadecimal, éste debe estar en forma de una cadena, ya que incluye letras y dígitos. Resulta que hay una relación bastante elemental entre los códigos ASCII de los números de Ø a 9, y los números mismos. Si suma 48 al número, obtendrá el código ASCII de ese dígito, que puede imprimirse como un carácter. Surge una ligera complicación cuando tenemos el diez, porque en hexadecimal es A, y el código ASCII de la A es 65, que es 55+1Ø. El programa de conversión, entonces, deberá añadir 48

Conversión: decimal-hexadecimal

(a) Octetos simples: números menores que 256 decimal.
Ejemplo: convertir 153 a hexadecimal.
153/16 = 9.5625; luego 9 es el dígito de mayor peso. El dígito de menor peso es Ø.5625*16 = 9. Luego, el número hexa completo es 99H.

Ejemplo: convertir 58 a hexadecimal.
58/16 = 3.625; luego 3 es el dígito más significativo. El dígito menos significativo es Ø.625*16 = 1Ø. Luego el número hexa completo es 3AH.

(b) Octetos dobles: números entre 256 y 65535
Ejemplo: convertir 23815 a hexadecimal.
23815/16 = 1488.4375; Ø.4375*16 = 7, dígito de menor peso.
1488/16 = 93 y resto Ø; luego Ø es el dígito siguiente.
93/16 = 5.8125; Ø.8125*16 = 13, que es D hexa. El último dígito es 5. Por tanto, el número hexa completo es 5DØ7H.

Fig. 3.5. Conversión decimal-hexadecimal para octetos simples y dobles.

unidades a los dígitos 9 ó menores, y 55 a los dígitos de diez a quince, para lograr una transformación correcta a código hexa. Esto no es difícil para un programa BASIC, y uno de tales programas para conversiones decimal-hexa, se presenta en la Figura 3.6.

Para realizar la transformación inversa, de hexa a decimal, se disponen las cosas de modo análogo, convirtiendo el código ASCII de cada dígito en el número correspondiente, multiplicando, además, por el factor de peso del dígito adecuado en cada caso (16,256 y 4Ø96); y, por último, sumando. La conversión puede usar un bucle para la multiplicación y la adición a la vez, y obtener el número decimal buscado. Una vez más, el programa BASIC es muy fácil (ver Figura 3.7) y, por eso, las páginas de las revistas especializadas en computadores personales están repletas de programas de conversión decimal-hexadecimal, para cada nuevo computador que aparece en el mercado.

A lo largo de este libro, utilizaremos principalmente códigos decimales, con unos cuantos valores hexadecimales mencionados donde sea preciso. Conviene señalar que existen aspectos de la programación en código máquina que exigen el dominio de ciertos métodos aritméticos. El más importante de ellos es la representación de enteros negativos.

```
10  CLS: PRINT "Introduzca el número decimal_":
    INPUT d
20  IF d>65535 THEN PRINT "Demasiado grande — el
    maximo es 65535": PAUSE 50 : GOTO 10
30  IF d<1 THEN PRINT "No valen los numeros menores
    que la unidad": PAUSE 50: GOTO 10
40  IF d< > INT d THEN PRINT "No se permiten
    decimales": PAUSE 50: GOTO 10
50  LET f=4096: LET h$=""
60  LET y=INT (d/f)
70  GO SUB 200
80  LET d=d-y*f : LET f=INT (f/16)
90  IF f<1 THEN GOTO 110
100 GOTO 60
110 FOR n=1 TO 3: IF h$ (1)="0" THEN LET h$
    =h$ (2 TO)
120 NEXT n
130 PRINT "El número hexadecimal es ";h$+"H"
140 GOTO 9999
200 IF y<=9 THEN LET h$=h$+CHR$ (y+48)
210 IF y>9 THEN LET h$=h$+CHR [&+55)
220 RETURN
```

Fig. 3.6. Programa BASIC de conversión decimal-hexadecimal.

Números negativos

En base diez, representamos los números negativos mediante el signo "menos", y así, podemos escribir valores como +15 y −15, que tienen los mismos dígitos, pero llevan un signo "+" ó "−" para indicar si son positivos o negativos. Los microprocesadores no cuentan con medios para representar dichos signos y, por lo tanto, el código binario debe emplear un bit para esa función. El bit que se elige siempre es el más significativo (el bit más a la izquierda). La convención que se sigue es que si el bit más significativo es un "1", entonces el signo del octeto es negativo, y si el bit más significativo es un "0", el signo del octeto es positivo. Es una convención muy simple y resulta muy útil, pero presenta desventajas para el operador humano. Una de esas desventajas es que los dígitos de los números negativos *no* son los mismos que los de sus correspondientes opuestos (de signo positivo): +5 en binario es 00000101, pero −5 es 11111011, que no se parece en nada al anterior.

```
1Ø CLS: LET y=1 : LET d=Ø : PRINT "Introduzca
   el numero hexadecimal" : INPUT h$
2Ø IF LEN h$>4 THEN PRINT "Demasiado grande - el
      máximo son" ' "cuatro caracteres": PAUSE 1ØØ :
      GOTO 1Ø
3Ø LET p$=h$ (LEN h$): LET h$=h$ (1 TO
   (LEN h$-1))
5Ø GO SUB 2ØØ : IF LEN h$>Ø THEN GOTO 3Ø
6Ø PRINT "El numero decimal es"; d
1ØØ GOTO 9999
2ØØ LET a=CODE p$
21Ø IF a<48 OR a>1Ø2 THEN GO SUB 3ØØ
22Ø IF a<65 AND a>57 THEN GOTO SUB 3ØØ
225 IF a1=97 AND a>7Ø THEN GO SUB 3ØØ
23Ø IF a<=57 THEN LET q=a-48
24Ø IF a>=65 THEN LET q=a-55
25Ø IF a>=97 THEN LET q=a-87
26Ø LET d=d+q*y : LET y = y*16
27Ø RETURN
3ØØ PRINT "Numero hexadecimal erroneo"'"Pruebe otra vez,
   por favor": PAUSE 1ØØ: RETURN
```

Fig. 3.7. Programa BASIC para conversión hexadecimal-decimal.

Una segunda desventaja es que, al tomar un bit para el signo, quedan menos bits para representar el valor del número. Si el bit de mayor peso de un octeto se dedica a la representación del signo, los siete bits restantes sólo pueden representar números hasta +127. No obstante, también podemos representar con ellos números negativos hasta −128, o sea, la cantidad de números diferentes que pueden representarse sigue siendo la misma. Si empleamos dos octetos, la pérdida del bit de signo significa pasar a tener un rango de −32768 á +32767, decimal. Si lo que manejamos son cinco octetos para cada número, como hace el Spectrum, un bit menos para el signo no afecta demasiado al conjunto de números representable.

La tercera desventaja es que los lectores humanos no pueden distinguir entre un número de un solo octeto que es negativo, y uno que se escribe sin tener en cuenta para nada los signos. La respuesta a esto último es que los humanos no tienen que preocuparse: el microprocesador opera con el número de igual modo, sin preocuparse de lo que éste represente. En este libro, pronto podrá ver cómo se efectúa la conversión y cómo se utiliza; con cierta práctica logrará dominar el método a la perfección.

En la mayor parte de las aplicaciones de interés para nosotros, los números negativos que aparecen son de un único octeto y, por ello, limitaremos nuestro estudio sobre el paso a forma negativa en ese rango. La conversión de los números binarios es el fundamento de todas las demás, y será la que presentemos en primer lugar, aunque posteriormente no la utilicemos demasiado. Para empezar, no podemos hacer negativos números de un solo octeto, cuyo valor decimal sea mayor que +127; tampoco podemos cambiar el signo de los números con valor decimal menor que −128, ya que ambas clases de números necesitan más de un octeto. La versión positiva del número, se escribe con ocho bits, y eso puede requerir que lo completemos con ceros a la izquierda. Si el bit más significativo no es cero, el número no puede transformarse en negativo.

Después se invierte cada bit, es decir, se cambia cada Ø por un 1, y cada 1 por un Ø, como ilustra la Figura 3.8. Al número resultante, denominado complemento del original, se le suma 1, y obtenemos la forma negativa o "complemento a dos", como también se le conoce, del número de partida. En la Figura 3.8 se observa el proceso de conversión, y se puede comprobar cómo el número binario con signo negativo, no se parece a su opuesto con signo positivo. Fíjese en la operación de suma binaria, donde 1 + Ø = 1, 1 + 1 = Ø y da un acarreo de 1, y 1 + 1 + acarreo de 1 = 1 y acarreo de 1. Traducido a base decimal, los números de 128 a 255 son negativos, y los comprendidos entre Ø y 127 son positivos. Para encontrar el valor equivalente de un número negativo en base diez, se resta, simplemente, el valor del número de 256 (vea los ejemplos de Figura 3.9). Esta es la manera más fácil de manejar los números negativos en el único sitio donde tendremos que hacerlo: en las instrucciones de salto relativo, que veremos más adelante. El quivalente hexadecimal de los números binarios negativos se puede obtener de su versión decimal o de la binaria.

Número binario	ØØ11Ø11Ø	54 decimal
Invertido	11ØØ1ØØ1	
Sumar 1	11ØØ1Ø1Ø	−54 decimal

Número decimal −5
En binario éste es 1Ø1, y en binario con ocho bits es ØØØØØ1Ø1
Invertido da..... 11111Ø1Ø
Sumando 1 ..... 11111Ø11 que es el octeto correspondiente a −5

Fig. 3.8. Cálculo del complemento a dos (forma negativa) de un número binario.

Número decimal -5	El octeto equivalente, en decimal,
es $256-5=251$	
Número decimal -8	El octeto equivalente, en decimal,
es $256-8=248$	

Fig. 3.9. Equivalente decimal de los números binarios negativos de un solo octeto.

Algunas instrucciones del microprocesador tratan cualquier número mayor que 127 decimal como un número negativo (7FH), y si se suman a otro número, el resultado es equivalente a restar ambos. En cambio, otras tratan todos los números como naturales (sin signo), lo cual significa que el bit de signo no tiene importancia como tal, y se cuenta como el número de veces que el octeto contiene a 128. El único problema surge cuando usted intenta averiguar qué ha hecho el microprocesador. En general, si en el juego de instrucciones lee que el número se supone con signo, quiere decir que el bit más significativo funciona como bit de signo. Si, por el contrario, el número se supone sin signo, será tratado simplemente como un número binario, y, en este caso, un octeto representa números positivos entre Ø y 255.

Capítulo 4

Descripción del Z-8Ø

Los registros: el contador de programa y el acumulador

Un microprocesador consiste en un conjunto de memorias, de un tipo bastante distinto a las memorias RAM y ROM, que se denominan *registros*. Estos registros están conectados entre sí y a los "pins" (recuerde que pins eran las "patitas" del circuito integrado) de la UCP, por medio de unos circuitos denominados puertas. En este capítulo, veremos algunos de los registros principales del Z-8Ø (idéntico a los Z-8ØA y Z-8ØB) y cómo se usan. Un buen punto de partida es el registro denominado* PC (Program Counter, en inglés) o Contador de Programa. El PC es un registro de dieciséis bits, que puede contener un número de dirección completo, hasta FFFFH ó 65535 decimal. Su función es contar los octetos de instrucción (¡no los programas, a pesar de su nombre!), de forma que el número almacenado en este registro se incrementará automáticamente en una unidad, cada vez que se lee un nuevo octeto de instrucción. Esa es una acción completamente interna que no hay que programar, porque está prevista en el modo de funcionamiento del Z-8Ø. El registro contador de programa (PC) empezará su cuenta desde Ø cada vez que se conecta el Z-8Ø y, por tanto, ésta será la dirección donde comienza la primera instrucción de la memoria ROM.

La utilidad del PC es que con él se direcciona la memoria. Cuando el PC contiene una dirección, las señales eléctricas que corresponden a los ceros y unos que forman el número de la misma, aparecen en un conjunto de conexiones, llamadas colectivamente bus de direcciones, que unen el microprocesador a toda la memoria, tanto RAM como ROM. Así pues, el número que está almacenado en el registro PC selecciona un octeto de la memoria, precisamente aquel que tiene esa

* A lo largo del libro mantendremos los nombres originales de los registros del Z-8Ø, pues con ellos aparecen en todas las referencias, traducidas o no, y cambiar ahora el nombre de los mismos por sus siglas en castellano, sólo podría acarrear una confusión enorme al lector. (N. de T.)

dirección. Al comenzar una instrucción, el microprocesador enviará una señal, llamada "*señal de lectura*", por otra línea, que hará que la memoria conecte sus bits almacenados a otro conjunto de líneas, el bus de datos. Las señales del bus de datos corresponden a la configuración de ceros y unos contenidos en el octeto de memoria a que ha sido seleccionado por la dirección existente en el PC. Cada vez que el número almacenado en el PC cambia, se selecciona otro octeto de la memoria, y es así como el microprocesador obtiene por sí mismo los octetos, incrementando el contenido del contador de programa cada vez que ha leído uno nuevo.

Hay otros medios para cambiar el contenido del PC, pero, de momento, los pasaremos por alto y estudiaremos otro registro: el acumulador. El acumulador es el principal registro "activo" de la UCP, lo que significa que, normalmente, se copia en él un octeto de memoria (carga del acumulador), o se utiliza para escribir en aquella (almacenar en memoria desde el acumulador).

Como su nombre sugiere, el acumulador guarda el resultado de las operaciones. Si tiene un octeto numérico almacenado en el acumulador, puede sumarle otro número, y el resultado quedará de nuevo en este registro. Es como si usted tuviese una variable numérica llamada Total, y escribiese la línea BASIC:

```
LET Total = Total + extra
```

donde extra es un número que se suma a Total. La diferencia, bastante importante, es que el acumulador no puede contener números mayores que 255, porque es un registro de un solo octeto.

El acumulador es muy importante, debido a que hay muchas más instrucciones para operar con él que con cualquiera de los demás registros del Z-8Ø. Cuando la UCP lee un octeto del PUERTO, éste se guarda en el acumulador, normalmente; las operaciones aritméticas se realizan corrientemente en él; cuando se escribe algo en la memoria, se suele hacer desde el acumulador, etc... A diferencia de los primeros microprocesadores que se fabricaron, el Z-8Ø tiene una gran cantidad de registros que pueden emplearse de forma muy similar al acumulador; pero ninguno de ellos tiene un rango tan amplio de instrucciones asociadas.

Modos de direccionamiento

Al escribir programas en BASIC, no tenemos que preocuparnos de las direcciones de memoria; de eso se ocupa el sistema operativo de la

ROM. Cuando se asigna un valor a una variable en BASIC con las líneas como:

LET n = 12

no nos importa saber dónde está almacenado el 12. De la misma manera, cada vez que escribimos:

2Ø LET k = n

no necesitamos conocer dónde está guardado el valor de n para copiarlo en k. Recordando nuestra comparación con la construcción de un muro, podemos esperar que en la creación de programas en código máquina tendremos que especificar cada número que utilicemos o, alternativamente, la dirección donde el número se encuentra almacenado. Esta última especificación es lo que se llama modo de direccionamiento, y es particularmente importante, porque se precisa un código diferente para cada modo de direccionamiento utilizado en cada uno de los comandos. Es decir: existen diferentes versiones de cada instrucción, y un código distinto para cada modo de direccionamiento posible. A estas alturas, una lista de todas las instrucciones del Z-8Ø sería bastante confusa para usted, y por eso se ha preferido incluirla en el Apéndice D. Lo que haremos a continuación es tratar algunos ejemplos sobre los modos de direccionamiento y la forma de indicarlos en lenguaje ensamblador.

Lenguaje ensamblador

Intentar escribir directamente código máquina como una ristra de números, es una labor difícil y que propicia los errores. La forma más útil de escribir un programa es disponerlo como una serie de pasos, en lo que se llama lenguaje ensamblador. Este lenguaje es un conjunto de abreviaturas de palabras, denominadas *mnemónicos,* y números que representan datos o direcciones. Dichos números pueden ser decimales o hexadecimales. Cada línea de programa en ensamblador, indica una instrucción del microprocesador y el conjunto de instrucciones abreviadas se traduce después a código máquina (a ese proceso también se le da el nombre de ensamblaje).

La misión de las líneas de un programa en ensamblador es presentar la instrucción y los datos o direcciones que se necesitan para ejecutarla, igual que cuando programamos un TAB en BASIC, debemos completarlo con un número. La parte del lenguaje ensamblador

que especifica lo que se hace, se denomina *operador,* y la parte que expresa sobre qué se realiza la acción, se llama *operando.* Como veremos después, algunas instrucciones no necesitan ningún operando. Pero la mayoría de las instrucciones del Z-8Ø constan de dos partes: una que especifica un registro, y otra que indica un octeto de datos o una dirección.

Un ejemplo permitirá seguirlo más fácilmente. Suponga que encontramos la línea de ensamblador:

LD A, 12

El operador LD es la abreviatura de la palabra LOAD (carga, en inglés), que se emplea para copiar o transferir un octeto desde un registro a alguna otra parte. "A" es la primera parte del operando y la abreviatura de Acumulador (en inglés se pone accumulator, y de ahí que coincidan las abreviaturas). Su posición inmediatamente detrás del operador indica que será el destino de un octeto, es decir, el sitio al que se transfiere el octeto al finalizar la instrucción. La segunda parte del operando, que viene tras de una coma, es el número 12. Por lo general, cuando los números se escriben como aquí, se suponen en base decimal, y cuando se manejan números hexadecimales, se expresa poniendo una "H" detrás (en este caso hubiera sido 12H).

La línea completa, entonces, tiene el efecto de colocar el número 12 en el registro acumulador del Z-8Ø. Es el equivalente en código máquina del comando BASIC:

LET a = 12

si imaginamos que la variable "a" es un circuito incorporado en la UCP, en vez de un grupo de posiciones de memoria, como sabemos que es.

Un comando de la forma LD A,12 se dice que utiliza *direccionamiento inmediato,* porque el octeto que se carga en el acumulador está colocado inmediatamente detrás del operador, en la propia instrucción. La parte de la línea LD A tiene sólo un octeto de código, que es 62 (3EH), de manera que la secuencia 62,12 en memoria, representará el comando completo LD A,12. Es mucho más sencillo acordarse del significado de LD A,12 que tratan de interpretar 62,12, y por esto utilizaremos el lenguaje ensamblador tanto como sea posible.

Este tipo de direccionamiento puede ser conveniente, pero exige poner un número explícitamente, lo mismo que en programas BASIC:

LET n = 4*12 + 3

en vez de utilizar:

LET n = a*b + c

En el primer caso, n siempre será 51, y podríamos haber escrito también:

LET n = 51

El segundo ejemplo es mucho más flexible, y el valor de n depende de lo que contengan las variables a, b y c. Cuando un programa en código máquina está en memoria RAM, los números que se cargan por medio del direccionamiento inmediato pueden cambiarse, si se desea hacerlo; pero si el programa se encuentra en memoria ROM, no es posible ningún cambio, y por eso necesitamos otros modos de direccionamiento. Uno de ellos es el *direccionamiento directo*.

El direccionamiento directo tiene una dirección de dos octetos como operando. Eso obliga al Z-8Ø a efectuar un gran trabajo, porque cuando ha leído el código de operación y la primera parte del operando (un octeto en total), tendrá que leer los dos octetos que están inmediatamente detrás del anterior, leer el octeto contenido en esa dirección y efectuar con él la operación de que se trate, y luego, incrementar el registro PC para que apunte a la siguiente instrucción (Figura 4.1). Por lo tanto, una operación directa es lenta, y precisa muchos octetos de memoria para contener la instrucción completa.

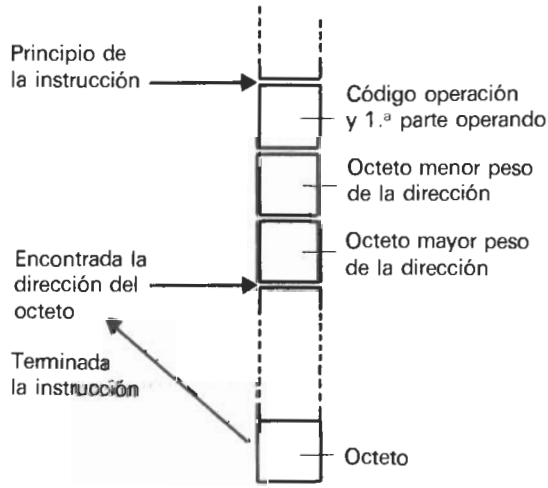


Fig. 4.1. Funcionamiento de una instrucción de almacenamiento con direccionamiento directo.

Suponga, por ejemplo, que tenemos la instrucción:

LD A, (7FFFH)

que aparece en la lista de operaciones como LD A, (NN), donde NN indica una dirección completa. En ensamblador, el operador es LD, carga, y el destino para el octeto es el acumulador A. El origen del octeto es la dirección 7FFFH (32767 decimal), y los paréntesis en este lenguaje significan "contenido de". Es un medio de recordarle que lo que se copia en el registro A no es 7FFFH (que no cabría en él), *sino el octeto que se encuentre almacenado en esa dirección*. El efecto de la instrucción completa es, pues, transferir el octeto contenido en la dirección 7FFFH al acumulador. Cuando se completa la instrucción, la dirección 7FFFH contendrá aún el mismo octeto, porque leer la memoria no altera para nada lo que hay almacenado en ella.

En ensamblador, los operandos se escriben de forma que el primero es el destino y el segundo el origen; o sea, si pusiésemos:

LD (7FFFH), A

estaríamos diciendo que el octeto guardado en el acumulador se copiase en la dirección 7FFFH. Observe que, de nuevo, se utilizan paréntesis. Algunos tipos de microprocesadores emplean la abreviatura ST (store, que significa "almacena" en inglés) para esta misma instrucción; en cambio el Z-8Ø, diferencia ambas acciones cambiando el orden de los dos operandos únicamente.

Direccionamiento indirecto

Los direccionamientos inmediato y directo (también conocido como extendido) son útiles, pero el Z-8Ø también permite una forma muy manejable del llamado *direccionamiento indirecto*. Este último consiste en ir a una dirección, mejor dicho a un par de ellas, y tomar de ella otra dirección, empleando la segunda para buscar o enviar el octeto de datos. Es algo parecido a ir a una agencia de viajes a que nos den la dirección del hotel en el que reservamos una habitación (o en el que comeremos, tal vez). El tipo de direccionamiento indirecto utilizado por el Z-8Ø se llama indirecto por registro, y en él intervienen algunos de los demás registros por parejas.

Emparejar dos registros de ocho bits para contener una dirección de dieciséis bits completa, es una característica especial del Z-8Ø, y una de las causas que han hecho este microprocesador tan popular entre los

diseñadores de computadores orientados a la gestión y otros usos serios. Hay tres grupos de tales registros que se denominan HL, BC y DE, respectivamente y, de ellos, el par HL es uno de los más utilizados para este fin. Cualquiera de los seis registros citados puede usarse separadamente, como si fuese otro acumulador de reserva, pero con un rango más limitado de operaciones.

Podemos cargar una dirección completa en el par de registros HL, por medio de un comando escrito en ensamblador de esta forma:

LD HL,32767 ó LD HL,7FFFH

Esta instrucción carga el octeto de mayor peso de la dirección, 7FH en hexadecimal, en el registro H (se llama así porque High significa alto en inglés: parte alta de la dirección), y el octeto de menor peso, FFH, en el registro L (Low = bajo, en inglés, parte baja de la dirección). Como puede deducir, los nombres de los registros se han puesto deliberadamente para recordarle (¡si es usted inglés, claro!) dónde se almacena cada octeto. Además, es posible copiar en el acumulador (o en cualquier otro registro) el octeto que se encuentra en la dirección 7FFFH, empleando el comando:

LD A, (HL)

o se puede transferir un octeto que esté en el acumulador a la dirección 7FFFH con:

LD (HL), A

Aquí tenemos otro ejemplo de cómo se utiliza el orden de escritura de los operandos para indicar cual es el origen y cual el destino; los paréntesis tienen su significado habitual, "contenido de".

Quizá piense usted que eso es una manera artificiosa de hacer lo mismo que con el comando LD A, (7FFFH), pero existe una importante diferencia. Una vez que se coloca en HL un número de dirección, podemos incrementar o decrementar el mismo con una instrucción de un octeto (sin operandos). Si hemos cargado la dirección 7FFFH en el par HL, entonces la instrucción:

DEC HL

tiene como resultado que el número almacenado en HL sea ahora 7FFEH (en decimal sería pasar de 32767 a 32766), de modo que usando:

LD A, (HL)

de nuevo, en el acumulador se copia el octeto que hay en 7FFEH (32.766), en vez del octeto almacenado en 7FFFH (32.767). Si está familiarizado con los bucles BASIC, se dará cuenta de cómo pueden emplearse las instrucciones del tipo anterior dentro de un bucle, para acceder a una dirección diferente cada vez que se repite dicho bucle, decrementando HL en cada iteración como en este ejemplo o, por supuesto, incrementándolo, si es eso lo que precisa.

Direccionamiento relativo del contador de programa

El direccionamiento relativo del PC es uno de los métodos de direccionar la memoria más simple y primitivo. El operando consiste en un octeto llamado desplazamiento, que se suma al contenido del PC del microprocesador, y el resultado es la dirección empleada para la carga, el almacenamiento o cualquier otra cosa que se esté efectuando.

Los primeros modelos de microprocesadores usaban este modo de direccionamiento prácticamente en todas sus acciones; pero el Z-8Ø únicamente lo emplea para un pequeño conjunto de instrucciones: las instrucciones de salto relativo (JR).

Un salto relativo es una transferencia a una nueva dirección, utilizando un método de direccionamiento relativo del PC. Las instrucciones JR completas constan de dos octetos: el operador JR y el operando, que, a su vez, comprende una condición y un desplazamiento. No obstante, estas instrucciones requieren un gran cuidado y experiencia en su manejo, puesto que el desplazamiento se trata como un octeto con signo. Esto quiere decir que si el bit de mayor peso del octeto es 1 (un valor de 128 o mayor, en decimal), entonces el desplazamiento se toma como negativo, y, cuando se sume al registro PC, la dirección resultante será *menor* que la que había en él antes de ejecutarse la instrucción JR. En términos de base decimal, pues, si el octeto es menor o igual que 127, se verificará un salto hacia delante; si el octeto está entre 128 y 255, lo que ocurrirá es un salto hacia atrás, en el programa. Una vez que ha tenido lugar el salto, el registro PC continuará su acción normal a partir de la nueva dirección, y no volverá a la dirección de partida, salvo por efecto del incremento automático de este registro (si fue un salto hacia atrás) o ejecutando otro salto (si el anterior fue hacia delante). Cuando se emplea un programa ensamblador para traducir el lenguaje ensamblador a código máquina, el propio traductor calcula el octeto de desplazamiento necesario en cada caso. En cambio, si es usted quien traduce "a mano", deberá calcular los desplazamientos por sí mismo. Para hacer el cálculo, el proceso que se sigue es de este tipo:

(1) Escribir la dirección en la que se colocará el octeto de operador de la instrucción JR. Esa será la dirección de origen.
(2) Escribir la dirección a la que el programa debe saltar, que será la de destino.
(3) Restar la dirección de origen de la de destino, y después restar dos al resultado obtenido. Lo que ha quedado es el valor del desplazamiento en decimal. Si es positivo, úselo directamente. Si es un número negativo, réstelo de 256, y emplee el resultado final como desplazamiento.

Dirección de origen 32542	
Dirección de destino 32565	La diferencia es 23
	Restamos 2 y obtenemos 21
	21 es el desplazamiento.
Dirección de origen 32533	
Dirección de destino 325Ø4	La diferencia es −29
	Restamos 2 y obtenemos −31
	El desplazamiento es
	256 − 31 = 225

Fig. 4.2. Desplazamientos positivo y negativo para una instrucción con direccionamiento relativo del contador de programa.

¿Por qué restamos dos? Se debe a que el desplazamiento se calcula siempre desde la dirección del operador, pero el salto no puede efectuarse hasta que se ha leído el octeto de operando, que es el que contiene el desplazamiento (lo que implica que el PC ya habrá sido incrementado), y además, el PC se incrementará automáticamente al final de la instrucción. Al restar 2, estamos teniendo eso en cuenta, y obtenemos el octeto de desplazamiento correcto. La Figura 4.2 da algunos ejemplos de desplazamientos positivos y negativos. Frecuentemente es más sencillo, cuando aún no ha decidido las direcciones finales, contar los octetos que hay entre los puntos de origen y destino. Observe que las direcciones relativas del PC no permiten saltos de más de 127 lugares hacia delante, o 128 lugares hacia atrás de la dirección donde se encuentra el octeto de operador, ya que ése es el rango que comprende un octeto con signo.

Los demás registros del Z-8Ø

Ya hemos mencionado una serie de registros del Z-8Ø. El PC es el registro de direccionamiento, que lleva la cuenta de los octetos de instrucción del programa (es el registro de "¿dónde estamos ahora?).

Cuando se ejecuta un programa en código máquina, la dirección del primer octeto del mismo debe colocarse en el PC. Posteriormente veremos cómo hacer eso en el Spectrum; pero, generalmente, no volvemos a cambiar explícitamente ese registro una vez que la ejecución ha comenzado.

El acumulador es el registro donde se efectúa la mayor parte del trabajo, y en el siguiente capítulo veremos con más detalle qué operaciones se pueden realizar con él. Existen seis registros más, de ocho bits, denominados B,C,D,E,H y L, que se pueden utilizar de forma similar al acumulador, aunque ninguno de ellos ofrece un rango tan amplio de operaciones. Además, como ya dijimos, esos registros pueden agruparse como BC, DE y HL, para almacenar números de 16 bits completos (las direcciones, por ejemplo). También se puede efectuar un número limitado de operaciones aritméticas de 16 bits en el par de registros HL.

Pero esto no cierra la lista de registros del Z-8Ø. Hay dos registros que nosotros utilizamos raras veces: el registro de interrupción (I) y el de refresco de memoria dinámica (R), cuya finalidad es muy específica. Otro registro de un solo octeto es el llamado *registro de estado o de indicadores*, que es muy importante, a pesar de que directamente no se pueda cargar nada en él, ni almacenarlo en memoria. El registro de estado es una forma de guardar indicaciones sobre el resultado de las operaciones. Cuando se lleva a cabo una operación como la suma, resta, o una operación lógica como AND, OR o XOR, el contenido del acumulador será un número positivo, negativo o cero. El "estado" positivo, negativo o cero se indica por medio del valor de los bits del *registro de estado*. Este nombre no es muy afortunado, porque no es, en realidad, un registro que pueda almacenar números, sino una colección de bits sin ninguna relación entre sí. Algunos libros denominan a este registro "*registro de indicadores*", ya que éste es un nombre descriptivo (se coloca un indicador en un estado cada vez que se efectúa un cálculo, y permanece en él hasta que se realice un nuevo cálculo).

En la Figura 4.3 aparece la disposición del registro de estado del Z-8Ø. Los bits que más nos interesan son S, Z y C, es decir, los bits 7, 6 y Ø respectivamente. El indicador S será 1 cuando el octeto contenido en el acumulador sea negativo, después de alguna operación aritmética/lógica; el indicador valdrá Ø, si el número almacenado en el acumulador es positivo o cero. El indicador Z es 1, cuando el acumulador contiene un cero tras efectuarse cualquier operación de las que alteran los indicadores, y será Ø en los demas casos. El indicador C será 1 cuando haya acarreo en una suma, o "toma" en una sustracción.

Las operaciones aritméticas y lógicas afectan a los indicadores, independientemente del registro en que se realizan. Es decir, si está

7	6	5	4	3	2	1	Ø Posición del bit
S	Z	X	H	X	P/V	N	C

INDICADORES

C—«1» si hay acarreo aritmético
N—«1» en operaciones de resta
Z—«1» si algunas operaciones dan cero
S—«1» si el resultado es negativo
(Los demás indicadores tienen un uso demasiado específico)

C: Acarreo
N: Suma/resta
P/V: Paridad/desborde
H: Acarreo medio
Z: Cero
S: Signo
X: No se utiliza

Fig. 4.3. El registro de estado del Z-80. Los bits 3 y 5 no se utilizan (pueden valer 1 ó 0).

trabajando con el registro C, y al hacer una resta el resultado es cero, el indicador Z toma el valor 1, al igual que si la operación se hubiese efectuado en el acumulador.

El programador, por lo general, no puede cambiar los indicadores del registro de estado, porque éste únicamente señala el resultado de las operaciones. Es más, el programador muy raras veces sabe directamente qué hay en el registro de estado, y su importancia radica en el hecho de que controla los saltos condicionales. Para hacer una comparación con BASIC, suponga que escribimos:

```
1ØØ IF a = Ø THEN GOTO 3ØØ
```

donde se efectuará un "salto" a la línea 3ØØ, si el valor asignado a la variable "a" resulta ser cero. No sabemos en qué instante ocurrirá esa circunstancia, sólo tenemos la certeza de que habrá un salto cuando "a" sea cero. La versión de esto en código máquina, con una instrucción de ensamblador, es:

```
JR Z, Despl
```

donde JR es el operador de salto relativo, Z es la parte del operando que especifica el indicador que se utiliza como condición de salto, y Despl es el octeto de desplazamiento, que dirigirá el salto a la dirección correcta, si éste tiene lugar. Cuando el microprocesador ejecuta esta instrucción, el octeto de código que representa la parte JR Z del comando, obliga a la máquina a comprobar el estado del indicador Z del registro de estado. Si la última operación aritmética/lógica dio un resultado cero (en el registro que se estuviese usando), el indicador Z valdrá "1", y el desplazamiento se sumará al contenido del PC, con lo

que se efectúa el salto. Si, en cambio, el indicador Z vale "Ø", el desplazamiento se ignora, de igual forma que en BASIC se pasan por alto las instrucciones que siguen a THEN, cuando no se cumple la condición. En nuestro ejemplo de BASIC, si la variable "a" no es cero, el programa no salta a la línea 3ØØ. En el caso del código máquina, la siguiente dirección que contendrá el PC será la que va inmediatamente detrás del octeto de desplazamiento (ver Figura 4.4).

Dirección - JR Z, despl

Indicador Z = "1", salto a la dirección resultante de: dirección de la instrucción JR + desplazamiento +2

Indicador Z = "Ø", se ignora el desplazamiento y se pasa a la instrucción que sigue a JR

Fig. 4.4. Instrucción de salto condicional.

Una peculiaridad de la forma en que el Z-8Ø trata el registro de estado, es que sólo ciertas acciones, particularmente las operaciones aritméticas y lógicas, alteran realmente los indicadores. Las operaciones de carga y almacenamiento no afectan a los indicadores, así es que si usted ha manejado antes el código máquina del microprocesador 65Ø2, tendrá que acostumbrarse a cosas como ésa. Por ejemplo, si tenemos un fragmento de programa en ensamblador así:

```
LD A, (HL)
DEC A
LD A, (DE)
JR Z, Despl
```

Si DEC A (decrementar el acumulador), que es una de las instrucciones que alteran los indicadores, hace que el contenido del acumulador pase a ser cero, al llegar a la instrucción JR Z, Despl, se efectuará el salto, incluso aunque el acumulador se haya cargado después con el octeto almacenado en la dirección que tiene el par DE, y probablemente ya no sea cero. Esto es una característica muy específica del Z-8Ø, y algunas veces resulta muy útil.

Registros índice

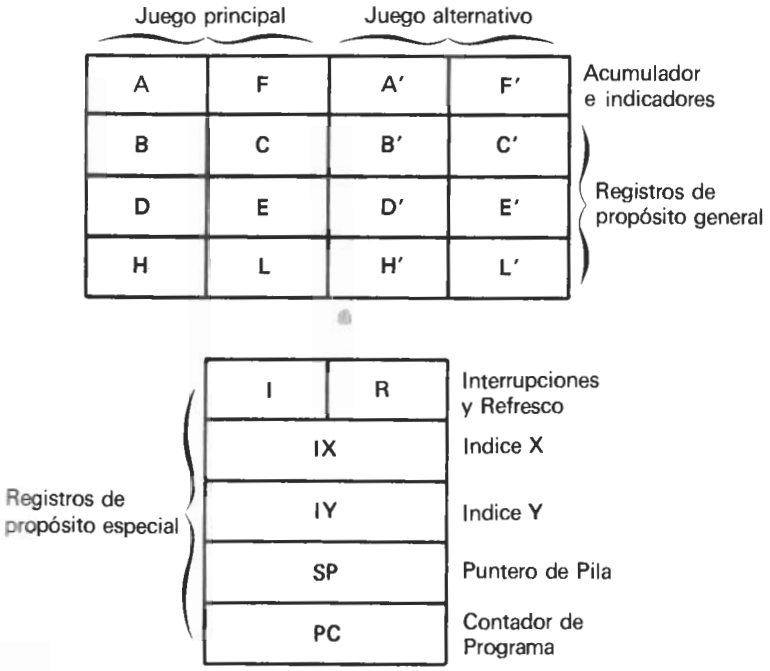
El Z-8Ø contiene también dos registros índice. Ambos son de 16 bits, y, por ello pueden contener direcciones completas. Sus nombres son IX é IY. Se denominan índices, porque pueden emplearse en una

forma de direccionamiento que es muy parecida al índice de los libros. Se almacena una dirección, llamada *de base* ó *dirección de página*, en uno de los registros índice, y, sumando un desplazamiento de un octeto al contenido del índice, se puede acceder a cualquier dirección hasta 127 octetos por encima, o 128 por debajo de la dirección de base. Esto se escribe en lenguaje ensamblador como (IX + d) ó (IY + d), donde "d" es el octeto de desplazamiento. Podemos utilizar de esta forma comandos como:

LD A, (IX + d)

que significa: "cargar el acumulador con el contenido de una dirección resultante de sumar el desplazamiento "d", a la dirección base almacenada en IX". El registro IX debería haberse cargado previamente con la dirección de base apropiada. No iremos más allá en esta breve introducción, porque los registros IX, IY se usan mucho en el sistema operativo del Spectrum, y el manual advierte que manejarlos en los programas de código máquina puede bloquear el computador.

En la Figura 4.5 tiene un "mapa" de los registros del Z-8Ø, que



Los juegos de registros principal y alternativo pueden intercambiarse con instrucciones tales como: EX, AF, AF' y EXX.

Fig. 4.5. El juego alternativo de registros no se puede usar al mismo tiempo que el juego principal.

permite una visión global de lo que hay disponible. Aparte de los registros principales A,F,B,C,D,E,H, y L, existe un "juego alternativo" de registros. Estos pueden utilizarse en lugar del juego principal, por medio de instrucciones de cambio. Pero no se pueden manejar a la vez ambos conjuntos de registros.

Capítulo 5

Operaciones con los registros

Operaciones con el acumulador

Puesto que el acumulador es el principal registro de un octeto, podemos enumerar las instrucciones que se refieren a él y describirlas en detalle, teniendo en cuenta que todo lo que digamos puede aplicarse al resto de los registros de 8 bits, cuando puedan utilizarse del mismo modo. De todas las operaciones con registros de un octeto, las transferencias son, por ahora, las más importantes. Por ejemplo, no podremos realizar ningún tipo de operaciones aritméticas con los códigos ASCII, y, por lo tanto, ese tipo de datos requiere principalmente instrucciones de carga del acumulador con el contenido de la dirección donde están almacenados, y un posterior almacenamiento desde el acumulador, en otra dirección de memoria distinta. La arquitectura interna del Z-8Ø no permite copiar un octeto de una posición de memoria a otra directamente, de modo que se emplea casi exclusivamente el "rudimentario" procedimiento de cargar un registro desde una posición de memoria, y después almacenar su contenido en otra distinta.

El siguiente grupo de instrucciones más importante es el aritmético y lógico; éste comprende la suma, la resta, AND (y-lógico), OR (o-inclusivo), XOR (o-exclusivo) y NOT (negación). Tenemos que añadir otras dos instrucciones al grupo: los desplazamientos, que consisten en mover los bits de un octeto una posición hacia la derecha o hacia la izquierda (el sentido del desplazamiento, además de otros detalles, depende del comando que emplee), y las rotaciones, que consisten en un desplazamiento de los bits del octeto, suponiendo que ambos extremos de éste estuviesen comunicados (ver Figura 5.1).

En la Figura 5.2 se muestran las diferentes acciones que tienen lugar cuando se ejecutan las principales instrucciones de rotación y desplazamiento. Después de efectuarse un desplazamiento, el registro pierde siempre uno de sus bits originales: aquel que se encontrase en el extremo hacia el que se desplaza y, para completar, se introduce un Ø en el extremo opuesto. En cambio, las rotaciones conservan los bits

originales del registro, pero alteran su posición dentro del octeto, respetando el orden relativo. Estas operaciones se utilizan, principalmente, para seleccionar la mitad de un octeto (o sea, un dígito hexadecimal) con propósitos aritméticos, y para enviar los bits del registro de uno en uno, o almacenarlos en él de la misma forma, en las operaciones de grabación y carga con el magnetofón. La Figura 5.3 resume el grupo de instrucciones aritméticas y lógicas, y sus mnemónicos en lenguaje ensamblador.

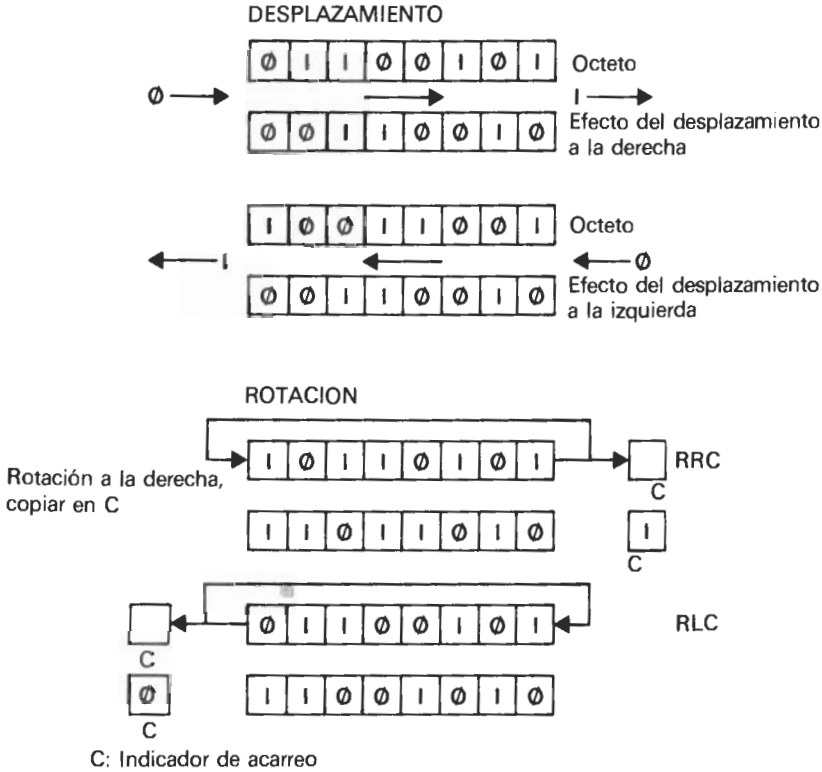


Fig. 5.1. Efecto de los comandos de Desplazamiento y Rotación (simplificados). Estos comandos del Z-80 a menudo utilizan, además de los contenidos de los registros, el indicador de acarreo.

Existe un tercer grupo que incluye las comparaciones, incrementos y decrementos. Incrementar y decrementar significa, respectivamente, sumar y restar una unidad; cuando se aplican a un registro simple, su efecto es incrementar o decrementar el número contenido en dicho registro. Si se trata de uno de los registros dobles (par de registros), las operaciones INC y DEC pueden utilizarse de dos maneras. Por ejem-

plo, el comando INC HL incrementará el número almacenado en el par HL, de forma idéntica a como INC A lo hace con el número que haya en el acumulador. Sin embargo, el comando INC (HL) incrementará el octeto contenido en la dirección de memoria que se encuentre en HL, sin alterar para nada la dirección en sí. Por ello, esa última podría simularse con:

```
LD A,(HL)   ;carga el octeto en el acumulador
INC A       ;incrementa el octeto
LD (HL),A   ;vuelve a almacenar el octeto en la misma dirección.
```

Observe cómo los comentarios en las sentencias de lenguaje ensamblador se ponen después de un punto y coma, análogamente a lo que ocurre con REM en BASIC (cualquier cosa que vaya detrás del punto y coma es un comentario, y no parte de la instrucción).

SLA Desplazamiento a la izquierda; el bit más significativo pasa al indicador de acarreo.

SRA Desplazamiento a la derecha; el bit más significativo no varía y el menos significativo pasa al indicador de acarreo.

SRL Desplazamiento a la derecha; pone Ø en el bit más significativo y el menos significativo pasa al indicador de acarreo.

RLD y RRD no se usan mucho.

RLCA Rotación izquierda del acumulador; el bit 7 se copia en el indicador de acarreo.

RLA Rotación izquierda del acumulador, incluyendo el indicador de acarreo.

RRCA Rotación derecha del acumulador; el bit menos significativo se copia en el indicador de acarreo.

RRA Rotación derecha del acumulador, incluyendo el indicador de acarreo.

RLC Rotación izquierda de registro; el bit más significativo se copia en el indicador de acarreo.

RL Rotación izquierda de registro, incluyendo el indicador de acarreo.

RRC Rotación derecha de registro; el bit menos significativo se copia en el indicador de acarreo.

RR Rotación derecha de registro, incluyendo el indicador de acarreo.

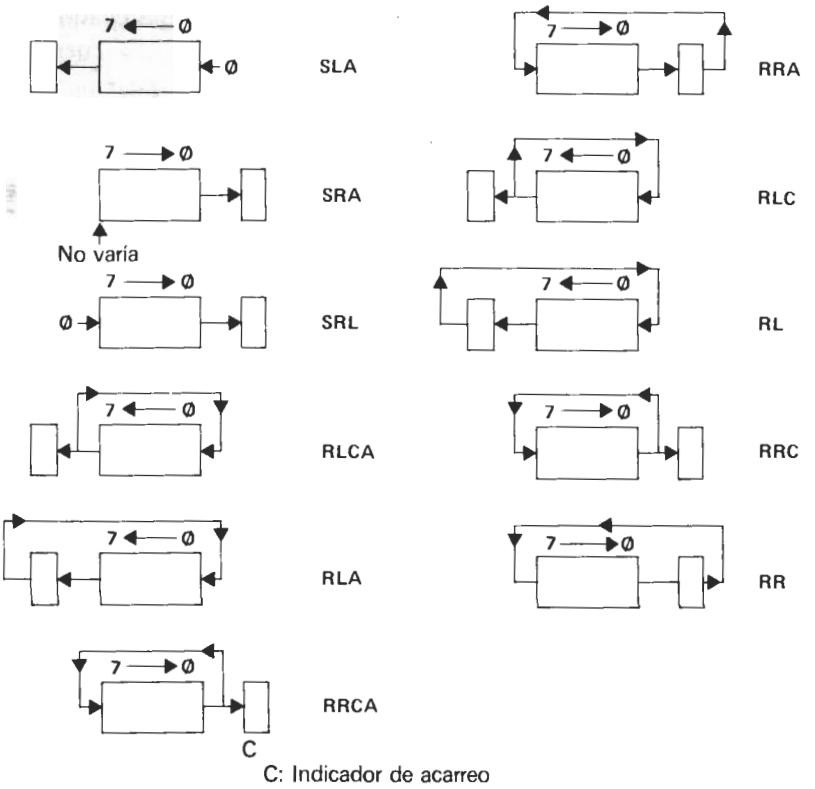


Fig. 5.2. Los principales comandos de Desplazamiento y Rotación del Z-80: su efecto y sus códigos mnemónicos.

Cuando se incrementa o decrementa un registro simple, el resultado alterará los indicadores del registro de estado, de forma que el indicador S tendrá valor "1", si el resultado es negativo, y cero en otro caso, y el indicador Z valdrá "1", si el resultado de la operación es cero. Lo mismo se aplica a las instrucciones INC(HL) o DEC(HL) respecto del octeto cuya dirección está en el par HL. Pero si se incrementa o decrementa un par de registros, con instrucciones como INC HL o DEC HL, entonces el resultado *no afecta* al registro de estado. Esta es otra peculiaridad del juego de instrucciones del Z-8Ø que a menudo resulta molesta, pero no hay nada que podamos hacer al respecto.

Más tarde veremos formas de programar que permiten colocar los indicadores de un modo determinado.

CP, el mnemónico de "compara", es una instrucción aritmética muy útil. CP debe ir seguido por un octeto, un registro o alguna dirección de origen de un octeto como (HL) o (dirección), y compara el octeto que especifique el operando (con uno de los modos que acabamos de citar) con el contenido del acumulador. La comparación es

ADD A,r	Sumar el contenido del registro r al del acumulador. Almacenar el resultado en el acumulador, y si hay acarreo, poner a "1" el indicador; ponerle a "Ø", si no hay acarreo.
ADC A,r	Sumar los contenidos del registro r y el acumulador, y sumar el valor del indicador de acarreo. Almacenar el resultado en el acumulador, y si hay otro acarreo, poner a "1" el indicador correspondiente, si no, ponerle a "Ø".
SUB r	Restar el contenido del registro r al del acumulador, almacenando el resultado en este último. Colocar a "1" el indicador de acarreo si hubo "toma", y a "Ø" si no hubo.
SBC A,r	Restar el contenido del registro r y el indicador de acarreo del contenido del acumulador, y poner el indicador de acarreo a "1" si hubo toma.
AND r	Calcular el "y" lógico de los octetos contenidos en el acumulador y el registro r. Poner el resultado en el acumulador. Los indicadores S y Z, principalmente, se ven afectados por la operación.
OR r	Calcular el "o-inclusivo" del acumulador y el registro r, almacenando el resultado en el acumulador. Afecta a los indicadores S y Z, principalmente.
XOR r	Calcular el "o-exclusivo" del registro r y el acumulador. El resultado se almacena en este último. Afecta, principalmente, a los indicadores S y Z.
ADD HL, rr	Sumar el contenido del par de registros rr al del par HL, y almacenar el resultado en el par HL. Colocar el indicador de acarreo a "1" si hubo acarreo desde el bit más significativo de H, y a "Ø" en otro caso.
ADC HL, rr	Sumar los pares de registros HL y rr, y sumar el bit de acarreo. El resultado se almacena en HL, y se pone el indicador de acarreo a "1" si hubo acarreo, y a "Ø" en caso contrario.
SBC HL, rr	Restar el contenido del par de registros rr, y el bit de acarreo al contenido del par HL. Almacenar el resultado en este último, y poner el indicador de acarreo a "1" ó "Ø", según hubiese toma o no, respectivamente.

NOTA: r indica cualquier registro, como B,C,D,E, etc., o un octeto cargado inmediatamente, o el contenido de una dirección que, a su vez, se encuentra en un par de registros como (HL).

No todos los modos de direccionamiento están disponibles para cada comando.

Fig. 5.3. Algunas instrucciones del grupo aritmético-lógico.

muy similar a una resta, pero hay una diferencia vital. Si, por ejemplo, teníamos el valor 5Ø en el acumulador, y ejecutamos SUB 4Ø, en el acumulador quedará ahora almacenado un 1Ø; con SUB 5Ø, en el acumulador hubiese quedado un Ø, y el indicador Z del registro de estado se hubiese puesto a "1". La instrucción CP 4Ø no haría nada apreciable, y CP 5Ø colocaría el indicador de resultado cero (Z) a "1"; ninguna de las dos instrucciones CP cambiaría el contenido del acumulador. Esta operación nos permite averiguar qué resultado tendría una sustracción, sin alterar el acumulador, y se puede emplear para colocar los indicadores antes de un salto condicional, cuando el valor del acumulador debe conservarse intacto, se produzca o no dicho salto.

Finalmente, tenemos las instrucciones de salto condicional. Se dividen en dos grupos: saltos absolutos y saltos relativos; los saltos absolutos se escriben en ensamblador como JP, y los saltos relativos como JR. Ambos tipos, a su vez, pueden ser incondicionales, o sea, el salto en cuestión se efectúa sin importar el contenido de los indicadores de estado. Alternativamente, pueden ser condicionales, lo que quiere decir que el correspondiente salto tendrá lugar solamente si el indicador seleccionado en el registro de estado tiene valor "1" ó "Ø", a elección del programador. Es muy parecido a la diferencia existente entre GOTO 3ØØ e IF A=Ø THEN GOTO 3ØØ (en BASIC). Cuando programamos saltos condicionales en ensamblador, tenemos que especificar cual es la condición, puesto que diferentes condiciones implican códigos de operación distintos. Por ejemplo:

JR Z, despl.

producirá el salto a la nueva dirección, si el indicador del resultado cero tiene valor "1", y la instrucción:

JR NZ, despl.

provoca un cambio de dirección, únicamente si el indicador de resultado cero *vale* "Ø". La lista completa de instrucciones de salto se explica brevemente en la Figura 5.4. Algunas de ellas se emplean muy raras veces en la clase de programas que, probablemente, escribirá usted en el Spectrum. Las órdenes JP necesitan ir seguidas de una dirección completa de dos octetos; los comandos JR llevan a continuación un octeto que indica el desplazamiento.

El código máquina en el Spectrum

Ha llegado el momento de comenzar a escribir algunos programas prácticos en código máquina para su Spectrum. No se trata de introducir líneas en lenguaje ensamblador directamente, porque, salvo si ha car-

JP dirección	Saltar a la dirección indicada.
JP c, direcc	Saltar a la dirección indicada, si se cumple la condición c.
JR despl	Salto relativo al contador de programa con desplazamiento "despl".
JR c, despl	Salto relativo al PC, con desplazamiento "despl", si se cumple la condición c.

Condiciones para JP	Condiciones para JR
NZ - distinto de cero	NZ - distinto de cero
Z - cero	Z - cero
NC - no acarreo	C - acarreo
C - acarreo	NC - no acarreo
PO - paridad impar	
PE - paridad par	
P - signo positivo	
M - signo negativo	

NOTA: Existe una versión de JP, JP (HL), que provoca un salto a la dirección contenida en el par de registros HL.

Fig. 5.4. Los comandos de salto relativos y absolutos, condicionales e incondicionales.

gado previamente un ensamblador (o sea, un traductor de ese lenguaje), el Spectrum ignorará "olímpicamente" tales comandos. Lo que hay que hacer es: encontrar los octetos de código máquina que corresponden a esas instrucciones de lenguaje ensamblador, escribirlas con POKE en la memoria de su computador, colocar la dirección del primer octeto en el registro PC del Z-8Ø A del Spectrum, y ver lo que ocurre después. Dicho así parece muy sencillo, pero cuesta un gran trabajo pensarlo, y hay que tomar una serie de precauciones. Para empezar, el Spectrum se reserva una considerable porción de memoria RAM, como hemos visto en los primeros capítulos del libro, para sus "operaciones auxiliares". Si ponemos en la memoria (con POKE) un conjunto de octetos, sin antes asegurarnos de lo que hacemos, lo más normal es que el propio Spectrum los reemplace con otros octetos correspondientes a la tabla de variables y otras informaciones internas, o que nuestros octetos borren algunos de los que el computador necesita para su correcto funcionamiento. Es posible preparar una sección de memoria para contener programas en código máquina de dos formas. Una es

introducir una sentencia REM como la primera de un programa BASIC, y rellenarla con espacios (tantos como octetos vaya a ocupar el programa de código máquina). Este era el método usado por el ZX-81, que no tenía ninguna otra previsión para programas en código máquina. El Spectrum, en cambio, permite reservar un fragmento de memoria, a través de la palabra clave CLEAR de BASIC. Si su máquina es un Spectrum de 16k, en el que la última dirección de RAM disponible es 32767 (decimal), las posiciones entre 326ØØ y 32767 están destinadas normalmente a los gráficos de usuario. Se puede destinar a sus propios programas de código máquina la zona justamente debajo de este área, empleando para ello el comando CLEAR. Así, CLEAR 325ØØ hará que la RAM existente entre las direcciones 325ØØ y 326ØØ se reserve para sus rutinas en código máquina, y si además no utiliza gráficos de usuario en el programa, nada le impide tomar con igual fin el espacio comprendido entre 326ØØ y 32767.

El siguiente problema consiste en colocar la dirección de comienzo de su programa de código máquina en el contador de programa (PC) del Z-80A. De nuevo, hay una sentencia BASIC que se encarga de ello: USR. Cuando la función USR va seguida de una dirección, el computador la introduce en el PC del Z-80A (y además en el par de registros BC), para que su programa se ejecute. Una vez que el Spectrum vuelve a su modo de funcionamiento normal, como ayuda extra dejará un número disponible: el contenido del par de registros BC al finalizar el código máquina. Si su programa empezó con una sentencia:

PRINT USR dirección

verá aparecer en la pantalla el contenido de BC. También puede asignar éste a una variable, escribiendo:

LET x = USR dirección

o asegurarse de que no se imprima ni asigne nada, si lo que usa es:

RANDOMIZE USR dirección

Observe que USR no puede escribirse solo, sino que debe aparecer detrás de alguna sentencia del tipo "haz algo": ¡es una función! Puede que usted no necesite o no quiera tener un número al retornar el control a BASIC, pero es muy útil. Si su programa no altera el par BC de registros, el valor de USR será la dirección que seguía a la función al ejecutarse la sentencia donde ésta figuraba (o sea, la de principio del código máquina).

El próximo paso es asegurarse de que el programa máquina termina de forma adecuada. Nada de lo que hemos visto hasta ahora indicará al Spectrum el final de una rutina en código máquina, por lo que el ordenador continuará leyendo octetos más allá del último de la rutina, hasta que encuentre alguno que provoque un "bloqueo" de todo el sistema. Esto se puede prevenir, poniendo como última instrucción de cualquier programa en código máquina la operación "vuelta de subrutina", que corresponde al código 2Ø1 decimal (C9H), cuyo efecto es volver automáticamente a BASIC cuando el programa empezó con un comando USR dirección.

Hay un punto del que todavía no nos hemos preocupado. Sus programas de lenguaje máquina se ejecutan en el mismo microprocesador Z-8ØA, que se encarga de todas las acciones del Spectrum. Si utilizamos los registros del Z-8ØA, debemos cuidar de no destruir información que necesite el Spectrum posteriormente. Por ejemplo, si en el momento en que comenzó su programa de código máquina, el Z-8ØA tenía en el par de registros BC la dirección del principio de la tabla de palabras clave (buscando USR quizá), no sería buena idea cambiarla por cualquier otra cosa. Cuando el código máquina se ejecuta por medio de la instrucción USR, ese problema lo resuelve el computador. Los contenidos de la mayoría de los registros se colocan en una parte reservada de la RAM que, frecuentemente, se denomina "pila". Este es otro de los motivos por el que hay que tener cuidado al introducir el programa en memoria; si usted borra cualquier cosa de la pila, ¡al Spectrum no le gustará nada! Una vez que se ejecute la instrucción RET, se restaurarán los valores de los registros automáticamente, y se volverá al funcionamiento normal en BASIC. En caso de que emplee otros medios para ceder el control a su programa, cosa que pueden lograr algunos ensambladores, la acción de salvar los registros tendrá que realizarla usted explícitamente. El mnemónico de lenguaje ensamblador que guarda el contenido de los registros en una pila, es PUSH, y los registros se almacenan por parejas, es decir, AF, BC, DE, HL, etc., en grupos de dos octetos. Para recuperar de nuevo los valores de la pila, hay que utilizar la instrucción POP, y los registros deberán restaurarse en orden correcto: "último en entrar, primero en salir" (es el método llamado LIFO), como los NEXT que siguen a los FOR en BASIC. Si escribió:

```
PUSH AF
PUSH BC
```

al comienzo de un programa, necesitará:

```
POP BC
POP AF
```

al final del mismo. Si pusiese:

```
POP AF
POP BC
```

¡habría intercambiado el contenido de AF y BC!. Esta propiedad es útil algunas veces (por ejemplo, para cambiar los valores del registro de estado), pero no es una técnica apropiada para un principiante.

Por el momento, nos olvidaremos de PUSH y POP, porque no se precisan cuando el programa se ejecuta por medio de USR. No obstante, hay una excepción. El Spectrum utiliza bastante los registros IX e IY, y éstos, al parecer, no son almacenados en la pila por el comando USR. Aún es factible manejarlos con seguridad, si se guardan en la pila antes de hacerlo, y posteriormente se restauran; de todos modos, es mejor no alterar sus valores, hasta que tenga alguna experiencia en programar el Spectrum con código máquina.

Programas prácticos

Ahora que hemos visto los preliminares, podemos empezar con algunos programas muy simples, destinados a familiarizarse con los métodos de colocar el código en la memoria del Spectrum, y con el uso del lenguaje ensamblador y del código máquina.

Como primer ejemplo, tomaremos el más sencillo posible, un programa que escriba un octeto en la RAM. En lenguaje ensamblador sería:

```
ORG 325ØØ     ;dirección de comienzo
LD A,85       ;cargar 85 inmediato
LD(3251Ø),A   ;ponerlo en 3251Ø
RET           ;volver a BASIC
```

La línea primera contiene un mnemónico, ORG, que no hemos visto antes y que, realmente, no forma parte del juego del Z-8Ø. Es una abreviatura de ORIGEN, y un recordatorio de que ésa es la dirección donde se encuentra el octeto inicial del programa, el primer octeto de la memoria reservada. Hemos elegido aquí una dirección que reserva mucho más espacio del que necesitamos, pero es tan buen ejemplo como cualquier otro, y deja mucho sitio libre para programas más extensos. Cuando se programa con un ensamblador, esta sentencia está permitida (ver Capítulo 8), y su efecto es que dicho ensamblador empiece a asignar direcciones, e incluso ponga los octetos de código de

operación, a partir de la posición especificada. Algunas veces esto se hace indirectamente, creando el código máquina en cinta o disco, en vez de directamente en la memoria.

El paso siguiente en la programación es escribir los códigos. Estos últimos deben buscarse cuidando de seleccionar los mnemónicos correctos. El código LD A,N, que es la forma en que aparece la carga inmediata en la lista, es 3EH o 62 decimal; por lo tanto, éste será el primer octeto del programa (a la sentencia ORG 325ØØ no le corresponde ningún código).

Podemos escribir una tabla de octetos y direcciones, empezando con:

325ØØ 62

El comando LD A,N debe ir seguido por el octeto de operando, el valor que se quiere colocar en el registro A, que es 85 decimal. Ahora la tabla presentaría este aspecto:

325ØØ 62
325Ø1 85

El siguiente octeto que necesitamos es el código de la instrucción LD (direcc.), A, que es 5Ø decimal (32H). Se pone debajo de los demás en la tabla, y tras él hay que especificar la dirección de almacenamiento deseada, en este caso 3251Ø. Aquí surge el inconveniente de convertir ese número en dos octetos y colocarlos en orden adecuado, es decir, primero el menos significativo. Encontrará que resulta más fácil si se dispone de una calculadora, o ¡se tiene conectado el Spectrum! Empleando la calculadora, se halla el valor de 3251Ø ÷ 256 y el resultado es 126.99218. Escriba debajo el 126, que será el Octeto de Mayor Peso, y póngale al lado OMP para recordarlo. Después lleve a cabo el cálculo siguiente (los símbolos con círculos indican las teclas de la calculadora que se pulsan):

126 × 256 = ± + 3251Ø =

Lo que está haciendo es multiplicar 126 por 256, obtener 32256 y restar esto de 3251Ø. El resultado final, 254, es el octeto de menor peso. Ya puede escribir la dirección 3251Ø como dos octetos: 254, 126 en su tabla, que aparece ahora así:

325ØØ	62
325Ø1	85
325Ø2	5Ø
325Ø3	254
325Ø4	126

Queda una última fila para la dirección 325Ø5, el octeto de retorno 2Ø1. Si no le ha gustado este procedimiento de calcular los octetos correspondientes a un número de dirección, use el programa para el Spectrum de la Figura 5.5. La próxima misión es colocar este corto programa en memoria. Debemos empezar reservándole espacio con CLEAR 325ØØ, y más tarde introducir con ayuda de POKE los octetos, de uno en uno. Puesto que el Spectrum, a diferencia del ZX-81, permite los comandos READ...DATA, el trabajo de escritura se simplifica mucho con un bucle que lea los octetos y los ponga en la dirección adecuada, empezando por 325ØØ y con tantas iteraciones como octetos tengamos. El programa sería:

```
1Ø CLEAR 325ØØ
2Ø FOR n=Ø TO 5: READ b
3Ø POKE 325ØØ+n,b
4Ø NEXT n
1ØØ DATA 62,85,5Ø,254,126,2Ø1
```

Observe cómo hemos puesto n=Ø TO 5, en vez de 1 TO 6, para que la primera dirección sea 325ØØ y no 325Ø1. Comience siempre a contar los octetos desde cero, no desde uno, y todo saldrá bien.

```
1Ø PRINT "Teclee la dirección" ' "Pulse Ø para
   terminar": INPUT d
2Ø IF d=Ø THEN GOTO 9999
3Ø IF d>65535 THEN PRINT "Demasiado grande-mayor
   que 65535": PAUSE 1ØØ : GOTO 1Ø
4Ø LET oms=INT (d/256) : LET omens=INT
   (d-256*oms)
5Ø PRINT "Los octetos de dirección son ";
   FLASH 1; omens; ","; oms
6Ø PRINT ' : GOTO 1Ø
```

Fig. 5.5. Programa BASIC para calcular los dos octetos correspondientes a una dirección.

Cuando ejecute este programa BASIC, parecerá que no ocurre nada. Todo lo que habrá hecho será colocar esos octetos en la memoria, y naturalmente, no pondrá en marcha el programa en código máquina. Antes de que hagamos eso, necesitamos algún método para comprobar que el código máquina se ejecuta realmente. Escriba PRINT PEEK 3251Ø y ENTER, y debería ver aparecer el resultado Ø,

a no ser que, por algún motivo, otro programa anterior haya almacenado algo en la posición 32510. Si acababa de conectar justo antes de introducir el código máquina, obligatoriamente el resultado tiene que ser cero.

Ahora teclee PRINT USR 32500 y ENTER. El número que se imprime en la pantalla es 32500. Es precisamente una señal de que el programa no altera el par de registros BC y permanece en ellos el valor de la dirección que seguía a USR, antes de comenzar a ejecutarse el código máquina, como ya habíamos comentado. Sin embargo, el Spectrum ha llevado a cabo la acción encomendada. Escriba el comando: PRINT PEEK 32510, y observará que el número 85 se imprime en pantalla. Este ha sido colocado en la posición 32510 por el pequeño fragmento de código máquina que introdujo.

No es más de lo que hubiese logrado desde BASIC con la orden POKE 32510,85, pero ya es un comienzo, y lo principal consiste en acostumbrarse a escribir programas en código máquina, almacenarlos en la memoria y ejecutarlos.

Pulsando NEW y ENTER, tendrá la pantalla borrada después de que aparezca el rectángulo negro; pero el programa en código máquina *no se ha borrado,* a pesar de que el programa BASIC, usado para escribirlo en la memoria, habrá desaparecido. Si escribe PRINT PEEK 32510, volverá a obtener el número 85, y si borra esa dirección de memoria con:

POKE 32510,0

seguido de ENTER, por supuesto (puede comprobar con otro PEEK que, efectivamente, ahora hay un cero en 32510), será posible ejecutar de nuevo el código máquina y cargar el valor 85 en la dirección 32510. Para hacerlo, teclee esta vez

LET x = USR 32500 (y luego ENTER)

En la pantalla no aparece nada al completarse esta clase de sentencia (aunque a la variable x se le habrá asignado 32500 y PRINT x lo pondrá de manifiesto), y por medio del comando directo PRINT PEEK 32510 verá, una vez más, escribirse el número 85. Para borrar la memoria completamente, puede usar:

CLEAR 32767 (en las máquinas de 16k)

y ENTER; seguido de NEW (ENTER). Eso borrará todo, lo cual debería recordar, porque si distraidamente intenta ejecutar la rutina en código máquina, con PRINT USR 32500, el Spectrum probablemente se bloqueará y no atenderá al teclado o ejecutará su rutina de iniciali-

zación. Esta situación es muy común cuando un programa en este lenguaje está equivocado, y el único medio es desconectar y volver a conectar el computador. Por supuesto, todo lo que hubiese en la memoria se perderá. A pesar de todo, vea el Capítulo 7, para saber cómo grabar en cinta el código máquina.

En raras ocasiones se necesita borrar la memoria de esta manera drástica. Si ha reservado espacio para un programa en código máquina por medio de CLEAR 325ØØ, puede escribir en esa zona tantos programas como desee. Si utiliza las mismas direcciones de nuevo, el programa más reciente reemplazará al que ya existía, pero mientras todos devuelvan el control a BASIC, ejecutando una instrucción RET (código 2Ø1 decimal), los octetos del programa anterior que permanezcan aún en sus lugares, no afectarán en nada al programa actual. Una cosa que hay que tener en cuenta, sin embargo, es cómo se emplea la memoria para almacenamiento, pensando otra vez lo que ya hicimos para poner el octeto 85 en la posición 3251Ø. Cuando las direcciones donde va a colocar datos de esa forma, están dentro del rango de direcciones donde se encuentra el propio programa, tendrá que escribir el programa auxiliar en BASIC, de tal modo que introduzca algunos octetos antes de las posiciones del dato, y otros después de la misma, pero ninguno en ella. No crea que porque usted no coloca nada allí, no existe nada. Puede hacer, si quiere, que el programa BASIC ponga un cero en ese espacio, pero no debe almacenar ninguna instrucción máquina en la dirección del dato. Tendrá que tener cuidado, también, de que la UCP no lea ese dato como si fuese un comando y, por tanto, tendrá que saltarse esa posición. La Figura 5.6 muestra cómo se logra eso, aunque para los principiantes es más seguro asignar una zona de almacenaje de datos separada del programa. Cualquier dirección por encima de la instrucción final (que tal vez será RET, o JP, o un salto relativo hacia atrás), resultará ser una buena elección. Algunas veces, será preciso no detallar los números concretos de dirección hasta saber cuántos octetos ocupará el programa, o elegir direcciones donde sea seguro que no existirá ninguna instrucción del mismo, como 32599 para un programa con un rango comprendido entre 325ØØ y 32599.

```
         LD A, (HL)        ;carga del acumulador
         JR 1              ;saltar el siguiente octeto
         (octeto de datos) ;octeto de datos usado en el programa
direcc.: LD C, A,          ;nueva carga del acumulador
         LD (direcc.), C   ;almacenar en la dirección del dato
```

Fig. 5.6. Salto sobre un octeto de datos. Esto debe hacerse cuando el dato está almacenado en medio del programa, pues la UCP no debe interpretarlo como una instrucción.

Ahora, intente otro ejemplo. Seremos más "atrevidos" esta vez, y volveremos a BASIC habiendo almacenado algo en el par BC de registros. Esto, recuerde, es simplemente una característica del sistema operativo del Spectrum, no del Z-8ØA; pero, puesto que estamos estudiando el código máquina para el Spectrum, no parece mala idea sacar partido de las ayudas que proporciona. La versión en lenguaje ensamblador está detallada en la Figura 5.7. No aparecen las direcciones asociadas, pero fíjese en que el comando RLA (rotación izquierda del acumulador) se compone de dos octetos, 2Ø3 y 23, por lo que el cargador BASIC tiene la forma:

```
 1Ø CLEAR 325ØØ
 2Ø FOR n=Ø TO 7: READ b
 3Ø POKE 325ØØ+n,b
 4Ø NEXT n
1ØØ DATA 62,85,203,23,6,Ø,79,2Ø1
```

```
LD A, 85      ;carga 85 en el acumulador
RL A          ;rotación a la izquierda
LD B, Ø       ;carga Ø en B
LD C, A       ;lleva el resultado a C
RET
```

Fig. 5.7. Programa en ensamblador que devuelve un octeto en el par de registros BC.

Como en la vez anterior, cuando se pulsa RUN, no ocurre aparentemente nada relevante, porque simplemente introduce los octetos en la memoria. Al escribir:

```
PRINT USR 325ØØ
```

y ENTER, el número 17Ø se imprime en la pantalla, y esto requiere una explicación. Lo que ha hecho el programa es cargar 85, que en binario es Ø1Ø1Ø1Ø1, en el acumulador. Una rotación izquierda de éste (Figura 5.8) da el número 1Ø1Ø1Ø1Ø binario,

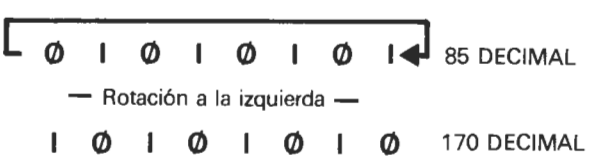


Fig. 5.8. Explicación del número que se obtiene.

que es el 17Ø decimal. Luego carga el registro B con cero, y efectúa una operación LD C,A para copiar en C el contenido del acumulador (17Ø), tras lo cual, el número 17Ø se encuentra en el par BC antes de volver a BASIC. Como el Spectrum siempre retorna con el valor almacenado en esa pareja de registros (menos cuando se usa RANDOMIZE 325ØØ), la parte PRINT del comando con el que llamamos a USR, se asegurará de mostrar dicho valor en pantalla. En el caso de haber usado LET x = USR 325ØØ, el valor asignado a x habría sido 17Ø.

Liste su cargador BASIC, y borre de la lista DATA los números 6 y Ø, teniendo cuidado de no dejar dos comas seguidas. Esa lista habrá quedado así:

```
1ØØ DATA 62,85,2Ø3,23,79,2Ø1
```

y deberá alterar la línea 2Ø, de forma que quede: FOR n = Ø TO 5. Ahora, ejecútelo para colocar los códigos en la RAM, y utilice de nuevo PRINT USR 325ØØ. Esta vez se imprime 32426, ¿por qué? La respuesta es sencilla. El sistema operativo del Spectrum pone la dirección que sigue a USR en el par BC. La dirección 325ØØ en forma de dos octetos en 126 (mayor peso) y 244 (menor peso), por lo cual, cuando se llama a USR 325ØØ, el contenido del registro B es 126, y el registro C contiene 244. El programa corregido omite el paso LD B,Ø. Entonces, el registro B tendrá 126 y en C estará el 17Ø, en el momento de volver a BASIC. Esos dos octetos forman el número decimal 256*126 + 17Ø = 32426.

Las acciones efectuadas automáticamente por el ordenador son muy útiles, pero siempre debe tener presente que los números almacenados previamente continuarán igual, salvo que se cambien explícitamente.

Capítulo 6

Diseño de programas

Los programas sencillos que hemos visto en el Capítulo V no hacen demasiadas cosas, aunque proporcionan cierta práctica muy útil en la conversión de programas en ensamblador a octetos de código, escritura de los mismos en la memoria y ejecución del programa resultante. En este capítulo trataremos el diseño de programas sencillos en código máquina o, en otras palabras, de cómo obtener la versión en lenguaje ensamblador de aquellos, puesto que hasta ahora ésta es, para el principiante, la parte más difícil de la construcción de programas en código máquina.

La dificultad, curiosamente, no surge porque el código máquina sea complicado, sino por su propia simplicidad, que impone el uso de gran número de instrucciones para conseguir cualquier cosa útil, y cuando un programa contiene muchos pasos es más difícil de planificar. La parte más compleja del planteamiento es subdividir lo que se quiere llevar a cabo, en un conjunto de etapas que las instrucciones del lenguaje ensamblador pueden completar. En esta parte del diseño, los organigramas son el método más apropiado para tener una visión global del problema. Yo nunca he creído que estos diagramas de flujo estén destinados a la planificación de programas BASIC; pero, en el caso del código máquina, pueden resultar realmente insustituibles.

Organigramas

Los organigramas son para los programas lo que los diagramas de bloques para los circuitos: presentan lo que se está haciendo (¡o intentando!) sin entrar en detalles más de lo necesario. Un organigrama se compone de un conjunto de símbolos que significan algún tipo de acción. En la Figura 6.1 aparecen algunos de los símbolos para organigramas más importantes, de cara al uso que haremos de ellos (tomados del conjunto normalizado en Gran Bretaña): el terminador (comienzo o fin), proceso (acción), entrada-salida y pasos de decisión.

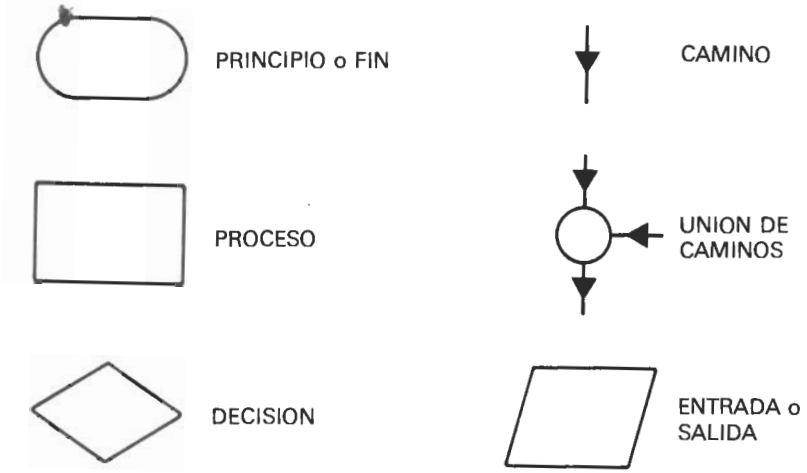


Fig. 6.1. Símbolos del organigrama. Una pequeña selección dentro de la gama normalizada británica.

Dentro de los símbolos escribiremos la acción que queramos, pero sin detallarla excesivamente.

La mejor manera de ver cómo se usan es siempre un ejemplo. Suponga que va a escribir un programa en código máquina, que tome el código ASCII de una tecla que se pulse, e imprima el carácter correspondiente a dicho código. Un organigrama para esa acción podría ser el de la Figura 6.2. El primer "terminador" es PRINCIPIO,

Fig. 6.2. Organigrama para el programa «obtener un carácter».

dado que todo programa tiene un comenzar en alguna parte. Este conduce al primer bloque de "acción", que está rotulado con "obtener carácter". Con esa frase se indica lo que queremos hacer (cómo lo haremos es algo que aún no sabemos). El siguiente bloque de "acción" es "ponerlo en BC", puesto que eso es lo que necesitamos para que el código ASCII sea devuelto a la rutina BASIC. A continuación viene el bloque "imprimir carácter", que es mejor efectuar en BASIC (en código máquina no sería tan directa la acción). El terminador FIN nos recuerda que el programa termina aquí, es decir, no es un bucle sin fin.

Este es un organigrama muy elemental (sin bucles ni decisiones), pero basta para ilustrar lo que queremos decir. Observe que las descripciones son bastante generales; por lo tanto, nunca ponga instrucciones de ensamblador en las "cajas" de acción de estos diagramas, porque eso conduciría a una confusión total. Estrictamente hablando, yo no hubiese escrito "ponerlo en BC" en un bloque; sin embargo, resulta tan esencial obtener un número al retornar a BASIC, que en esta ocasión es preciso recordarlo explícitamente. Los diagramas de flujo (el otro nombre de los organigramas) no deberían ser algo que sólo el propio diseñador entienda, sino una forma clara de expresar lo que se va a realizar, comprensible para cualquiera. Desgraciadamente, muchas veces se dibujan los organigramas después de haber escrito los propios programas como medio de aclarar las acciones que ejecutan aquellos. No cometa usted este error.

Una vez que disponemos del organigrama, podemos comprobar que éste responde verdaderamente a lo que queremos hacer, siguiéndolo cuidadosamente de principio a fin. En el ejemplo de la Figura 6.2, las partes "obtener carácter" y "ponerlo en BC" se van a escribir en código máquina, así es que nos centraremos en ellas por ahora.

Obtener el código ASCII de una tecla pulsada, parece algo enrevesado al principio. Muchos computadores ponen dichos códigos en el acumulador, mediante las subrutinas de lectura del teclado, y después guardan temporalmente el valor en la memoria RAM. Según la descripción de las variables del sistema que da el Capítulo 25 del manual, la posición 23560 se destina a esa misión: contener el código de la última tecla pulsada. Si cargamos A con el contenido de esa dirección, tendremos en el acumulador el código ASCII de la última tecla que se pulsó; el paso 1 está terminado. Luego viene algo ya familiar: queremos tener 0 en el registro B, y copiar el contenido del acumulador en el registro C. Hemos de hacerlo así, porque no es posible cargar C desde la memoria directamente (ésa es una de las restricciones que hace del registro A el más útil del conjunto de un solo octeto). Con el par BC cargado, podemos retornar a BASIC y utilizar el valor almacenado en dicho par, según nos interese. Ejecutando el código máquina con LET

x = USR 32500, bastará imprimir CHR$ x en pantalla para tener el carácter deseado.

Tenemos ya resuelta la mitad del problema. Lo siguiente que se plantea es cómo obtener el octeto de código ASCII desde BASIC. No sirve la sentencia INPUT, porque dejará siempre un 13 almacenado en esa posición, ya que ése es el código ASCII de ENTER, que es la última tecla que debe pulsarse para finalizar el comando. Seguramente será más útil INKEY$; podemos establecer un bucle del cual se salga sólo cuando se haya detectado la pulsación de una tecla y, tras él, una llamada a la rutina en código máquina, y que puede ya leer el código de la tecla en la dirección 23560.

Se comienza diseñando el programa en lenguaje ensamblador, que podría ser:

```
LD B,0          ;borrar B
LD A,(23560)    ;obtener el código ASCII
LD C,A          ;ponerlo en C
RET             ;retorno a BASIC
```

A primera vista, parece que haría bien lo que queremos. Ahora crearemos un programa en BASIC para llamar al anterior:

```
200 LET t$ = INKEY$ : IF t$ = "" THEN GOTO 200
210 LET x = USR 32500
220 PRINT CHR$ x
```

Por último, completamos todas las etapas traduciendo el programa ensamblador a código máquina (por medio de las tablas de instrucciones) y escribimos la parte de programa BASIC encargada de introducir el código en la memoria:

```
10 CLEAR 32500
20 FOR n = 0 TO 6: READ b
30 POKE 32500 + n, b
40 NEXT n
100 DATA 6,0,58,8,92,79,201
```

Añada esto a las líneas de 200 a 220 y quedará todo listo. Con toda seguridad, la ejecución del programa completo escribirá en la pantalla el carácter de la tecla que pulse. De nuevo, no parece muy interesante, porque lo mismo se podría conseguir con:

```
200 LET t$ = INKEY$ : IF t$ = "" THEN GOTO 200
220 PRINT t$
```

La intención, sin embargo, es adquirir experiencia en la escritura de código máquina y aprender cómo relacionar ese código con el BASIC del Spectrum; ¡nadie pretende empezar escribiendo trabajos geniales!

Vamos a ver algunas formas de eliminar ciertos defectos de este programa. En primer lugar, las dos líneas 21Ø y 22Ø pueden sustituirse por la línea única:

```
PRINT CHR$ USR 325ØØ
```

Esta última puede interpretarse así: "imprimir el carácter cuyo código es obtenido por la subrutina USR 325ØØ". Después, podemos preguntarnos acerca de INKEY$. ¿Necesitamos emplear BASIC aquí? La contestación es negativa; con el programa de la Figura 6.3 creamos nuestro propio bucle INKEY$. La sección de código máquina coloca

```
LD A, (2356Ø) ;obtener el último carácter pulsado
LD B,Ø        ;borrar B
LD C,A        ;cargar en C el resultado
RET           ;regreso a BASIC

1Ø CLEAR 325ØØ
2Ø FOR n=Ø TO 6 : READ b
3Ø POKE 325ØØ+n, b: NEXT n
5Ø DATA 58,8,92,6,Ø,79,2Ø1
1ØØ LET x= USR 325ØØ : IF x<=32 THEN GOTO 1ØØ
11Ø PRINT CHR$ x
```

Fig. 6.3. Programa para leer un carácter. Se necesita una parte BASIC para colocar el código en memoria y comprobar si se ha pulsado alguna tecla.

el contenido de la dirección 2356Ø en el par BC de registros, como antes; la parte BASIC comprueba ese octeto y, si es menor que 32 (el código de "espacio"), se repite el bucle hasta que se obtenga un valor mayor que 32. Eso no nos libera por completo de BASIC, debido a que todavía dependemos de la línea 1ØØ para la comprobación. ¿Podríamos comprobar el código y volver a un bucle dentro del propio programa en código máquina? ¡La respuesta *por el momento* es que no! Para almacenar un nuevo octeto en la posición 2356Ø, el sistema operativo del Spectrum tiene que estar ejecutándose, cosa que será imposible mientras nuestro propio programa esté continuamente en un bucle, leyendo el contenido de esa dirección. Si deseamos comprobar las teclas que se pulsan, es necesario buscar otros métodos, como veremos posteriormente.

Nos conformaremos, a estas alturas, con examinar otro aspecto de la entrada. Supongamos que nos hace falta cambio a letras mayúsculas automático (o sea, una tecla SHIFT automática), cuyo propósito es imprimir "H" cuando se apriete la tecla "h". Observando el código ASCII, descubrimos que los códigos de las minúsculas son 32 unidades mayores que las de las correspondientes mayúsculas. En consecuencia, podríamos intentar restar 32 de los códigos de esas letras minúsculas, y lograr así las mayúsculas. Hay que tener prevista la circunstancia de que se pulsen directamente las letras mayúsculas, y eso se logra impidiendo que se verifique el cambio, si el código ASCII del último carácter tecleado es menor o igual que 96, puesto que el código de "a" es el número 97.

Un organigrama de tareas, muy simple, podría ser el de la Fig. 6.4.

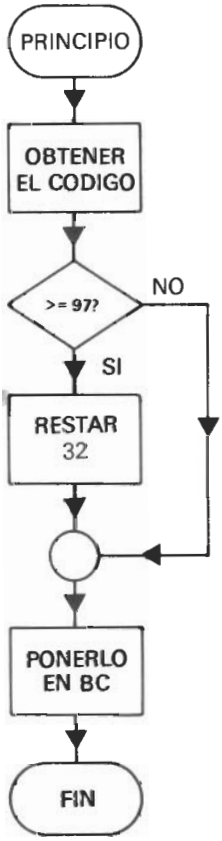


Fig. 6.4. Organigrama del programa de conversión a mayúsculas.

Queremos leer el código y compararlo con 97. Si es mayor o igual que 97, restaremos 32; pero en el caso de ser menor que el número citado, lo dejaremos como está. En ambos casos, cargamos el resultado en BC y ahí termina la sección de código máquina.

Hemos introducido un paso de decisión que hasta ahora no habíamos usado en lenguaje ensamblador. Por ello seguiremos con más detalle las próximas etapas. La Figura 6.5 muestra la versión en ensamblador del organigrama. Se obtiene el código del carácter, como

```
        LD A, (23560)      ;cargar el carácter en A
        CP 97              ;comparación
        JR C, salida       ;salir si es menor que 97
        SUB 32             ;en otro caso restar 32
Salida: LD B, 0            ;borrar B
        LD C, A            ;cargar respuesta en C
        RET                ;volver a BASIC
```

Fig. 6.5. Versión en lenguaje ensamblador del programa de conversión.

ya vimos, cargando el acumulador con el contenido de la dirección 23560. Para el paso de decisión se precisan dos instrucciones. La primera es CP 97, cuyo significado es comparar lo que haya en el acumulador con 97. Como dijimos en su momento, es muy parecido a una resta, pero sin que varíe el octeto almacenado en el acumulador. Si éste es mayor que 97, se le resta 32; en caso contrario, es menor que 97, la resta de la operación CP causará una "toma" (acarreo al restar), lo cual quedará reflejado en el indicador de acarreo del registro de estado, que tendrá valor "1". Como resultado de ello, se ejecuta el salto a "salida", y el octeto queda inalterado. La palabra "salida" es una etiqueta, que sirve en lenguaje ensamblador para indicar la dirección de salto de manera simbólica, evitando tener que calcular en esta etapa desplazamientos y direcciones numéricas. El punto de destino se confirma poniendo la etiqueta del mismo nombre que la figura, en la instrucción de salto a la izquierda del paso donde debe seguir la ejecución al verificarse dicho salto, seguida de dos puntos (en este ejemplo, la etiqueta se escribe junto al comando LD B,0, punto de destino para JR C, salida).

El resto de las instrucciones, a partir de "salida", es idéntico al caso que examinamos anteriormente. Una minuciosa revisión del programa en ensamblador muestra que éste debería hacer correctamente lo que se pretende, o, de otro modo: sigue las acciones del organigrama. Es el momento, entonces, de pasar a la codificación. Esta resulta bastante directa, exceptuando el caso de la orden de salto. La diferencia entre

las direcciones de JR y LD B,Ø es de 4 octetos, lo que nos permite calcular el desplazamiento correcto con las reglas dadas en el Capítulo 5; o sea, restamos 2 y conseguimos el octeto de desplazamiento final que es 2. El programa definitivo se presenta en la Figura 6.6. Existe, en ese programa, una línea inesperada. Si omitimos las sentencias que vienen después de la 12Ø, todo funcionará aún adecuadamente, pero se

```
 1Ø CLEAR 325ØØ
 2Ø FOR n=Ø TO 12
 3Ø READ b: POKE 325ØØ+n, b
 4Ø NEXT n
 5Ø DATA 58, 8, 92, 254, 97, 56, 2, 214, 32,
    6, Ø, 79, 2Ø1
1ØØ LET k$=INKEY$: IF k$=""" OR CODE k$=
    13 THEN GOTO 1ØØ
11Ø LET x=USR 325Ø
12Ø PRINT CHR$ x;
13Ø IF INKEY <> """ THEN GOTO 13Ø
14Ø GOTO 1ØØ
```

Fig. 6.6. Versión codificada del programa de conversión.

imprimirá una única letra en cada ejecución, al pulsar alguna tecla. Es mucho más práctico añadir un bucle para que el programa se repita varias veces; no bastaría, sin embargo, con introducir solamente un GOTO 1ØØ, porque encontraríamos efectos curiosos, tales como letras repetidas o espacios no deseados. El motivo de ello es que la memoria tampón (buffer, en inglés) asociada a la función INKEY$, no se borra siempre a tiempo. Con la línea 13Ø se espera hasta que INKEY$ esté en blanco. Comprobará que este programa da una letra mayúscula como salida, cada vez que apriete alguna tecla literal. Tampoco esta vez hemos creado una obra maestra (¿qué pasa si se pulsa una tecla numérica?). Hubiese sido posible hacer lo mismo completamente en BASIC, usando algunas de las direcciones de la memoria RAM reservada; pruebe el programa de la Figura 6.7, y verá cómo la mayor parte de las teclas dan el resultado correcto, aunque algunas imprimen un signo de interrogación (lo que significa que el número almacenado en esa dirección no es un código ASCII). Repetimos que el propósito de los ejemplos es el aprendizaje de los métodos de programar en lenguaje máquina; por otra parte, si emprendiésemos la escritura de rutinas que efectúan trabajos imposibles de lograr con BASIC, terminaríamos teniendo ejemplos que le resultarían muy difíciles de comprender en esta etapa.

```
1Ø LET k$ = INKEY$: IF k$ = "" OR CODE k$ =
   13 THEN GOTO 1Ø
2Ø LET x = CODE k$
3Ø IF x> =97 THEN LET x = x – 32
4Ø PRINT CHR$x;
5Ø IF INKEY$ <> "" THEN GOTO 5Ø
6Ø GOTO 1Ø
```

Fig. 6.7. El programa de conversión en BASIC.

En cualquier caso, ya hemos manejado un paso de decisión en un organigrama, en un programa de lenguaje ensamblador y en la traducción de éste a código máquina. Es un paso más en nuestro camino, e incrementa nuestra experiencia en la carga y ejecución de código máquina. Quizás podríamos intentar en este momento algo más ambicioso, y, al mismo tiempo, investigar los secretos del Spectrum con mayor profundidad. Anteriormente descubrimos que existían ciertas dificultades asociadas a la simulación del funcionamiento de INKEY$ sin utilizar BASIC. Vamos a ver ahora la forma de hacerlo. Hay una rutina, dentro de la ROM, para leer el teclado, y, si la encontramos, podremos utilizarla en nuestros propios programas. Con ayuda del desensamblador de Campbell no fue difícil dar con ella. Busqué un segmento de programa que cargara la dirección 2356Ø y, una vez detectado, fui hacia atrás en pos del punto donde empezaba. Esto me llevó a la dirección Ø2BFH, 7Ø3 decimal, a partir de la cual comienza una subrutina que lee las señales eléctricas del teclado y las transforma en códigos ASCII. Una ejecución preliminar de la misma, reveló que colocaba 255 en el acumulador, si no se pulsaban las teclas, o el código ASCII de una tecla que se hubiese apretado en el teclado.

El empleo de una rutina ROM del Spectrum, es una novedad respecto de lo que hemos visto hasta el momento. Las subrutinas en BASIC se llaman por medio de la sentencia GOSUB, y el retorno de la misma se efectúa cuando se ejecuta el comando RETURN al final de la subrutina. En lenguaje ensamblador, una subrutina se llama con la instrucción CALL, seguida de la dirección de comienzo de la misma. El retorno se logra con la instrucción RET, que ya hemos estudiado. CALL y RET tienen que aparecer juntas, y la razón de que podamos volver al funcionamiento en BASIC con la misma orden RET, es que la correspondiente orden CALL está incluida en la rutina que interpreta la función USR de BASIC.

Por todo lo dicho, si escribimos la instrucción CALL 7Ø3, obtenemos en el acumulador el código de la tecla pulsada (si es que se pulsa alguna). El código de operación para CALL es 2Ø5, y la dirección que

le sigue se escribe en forma de dos octetos, como ya es habitual: el de menor peso primero y después el de mayor peso (CALL debe ir seguida de dos octetos, aunque uno de ellos sea cero). Los dos octetos que corresponden a 7Ø3 son 191,2; luego el conjunto de números 2Ø5,191,2, ejecutará la acción CALL 7Ø3.

La Figura 6.8 ilustra el organigrama que precisaremos para este programa. Se busca el octeto del teclado por medio de la instrucción CALL, y se comprueba si es 255, puesto que ese número es el que pone la rutina ROM en el acumulador, si el teclado no se toca (es el número que significa "falso"). Si el octeto es igual a 255, se repite la llamada; pero si se ha apretado una tecla, el contenido del acumulador será el código ASCII de aquella. Ese valor del acumulador se pasa entonces al par BC de registros, como hacíamos antes.

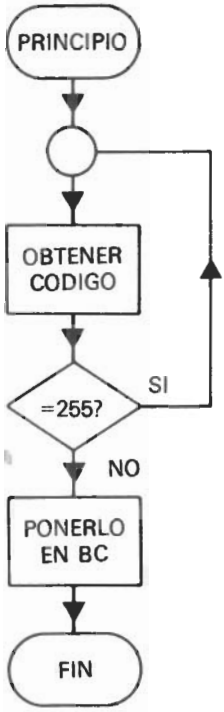


Fig. 6.8. Organigrama del programa para leer una tecla.

El programa BASIC de la Figura 6.9 introduce en la memoria el código y hace uso de él. Para variar, he incluido un método más rápido de manejar los comandos POKE y USR. Situando LET j = 325ØØ cerca del principio de programa, podemos ahorrar memoria escribiendo j en lugar de 325ØØ.

```
 1Ø CLEAR 325ØØ
 2Ø LET j=325ØØ
 3Ø FOR n=Ø TO 1Ø
 4Ø READ b: POKE j+n, b: NEXT n
1ØØ DATA 2Ø5, 191, 2, 254, 255, 4Ø, 249, 6, Ø,
     79, 2Ø1
2ØØ PRINT CHR$ USR j
```

```
Lenguaje ensamblador:        Atrás: CALL Ø2BFH
                                    CP 255
                                    JR Z, Atrás
                                    LD B, Ø
                                    LD C, A
                                    RET
```

Fig. 6.9. Programa BASIC para almacenar el código máquina en memoria y utilizarlo.

Además, dado que eso evita una conversión de ASCII a binario cada vez que se pone 325ØØ, también ahorramos tiempo. El programa hace en código máquina lo mismo que INKEY$ en BASIC; espera a que se pulse una tecla, y luego retorna con el valor del código ASCII en el acumulador.

Esta es la primera vez que usamos una rutina de la ROM del Spectrum, cosa que no siempre es tan sencillo de hacer. A menos que usted disponga de un desensamblador, de mucho tiempo libre y de un gran dominio del lenguaje ensamblador, resulta bastante complicado descubrir rutinas existentes en memoria ROM (fue un golpe de suerte que la rutina para INKEY$ se pudiese encontrar tan fácilmente). No obstante, hay libros con la memoria ROM del ZX-81 totalmente desensamblada y comentada, y será cuestión de tiempo que salgan al mercado tratamientos análogos para el Spectrum* (algunas de las rutinas son casi idénticas, aunque a veces ocupan direcciones diferentes). Cuando disponga de ese tipo de material, podrá buscar las subrutinas por sí mismo y emplearlas en los programas; pero debe tener precaución, y asegurarse de que el contenido de los registros es apropiado, antes de llamarlos.

Ha llegado la hora de que intente algo por sí solo: ¿sería capaz de combinar la rutina que acabamos de examinar, el equivalente en código máquina de INKEY$, con la idea de restar 32, si el octeto es mayor o igual que 97, para crear un programa que imprima las teclas pulsadas siempre en mayúscula?

---

* En el momento de escribirse esta traducción, existen ya tales libros en nuestro país, aunque aún no tengo noticias de ninguno en nuestro idioma. *(N. de T.)*

Bucles

La forma de crear bucles que más se utiliza en el Spectrum es el par de comandos FOR...NEXT. Estos tienen una variable "contador" para saber cuántas veces se ha repetido el bucle, y en cada paso por el mismo compara el valor del contador con los límites que se hayan establecido. La Figura 6.10 muestra cómo puede simularse la acción de FOR...NEXT, sin escribir ambos explícitamente, dentro de un programa BASIC. Para ello, se incluye una sentencia IF...THEN, que decide si el bucle debe continuar o no. Casi todo los microprocesadores permiten programas los bucles de un modo similar, utilizando un registro o una posición de memoria como contador. La familia Z-8Ø posee, además, un grupo de instrucciones de bucle predefinidas, que ahorran un considerable esfuerzo de programación.

```
1Ø LET contador = Ø: LET límite = 1Ø
2Ø PRINT "Paso"; contador
3Ø LET contador = contador + 1
4Ø IF contador < = límite THEN GOTO 2Ø
5Ø PRINT "Terminado"
```

Fig. 6.10. Cómo puede simularse un bucle FOR...NEXT en BASIC.

El equivalente de FOR...NEXT en el Z-8Ø se escribe en lenguaje ensamblador como DJNZ, abreviatura (inglesa ¡claro!) de "decrementa y salta si no es cero". El registro B es el contador, y cada vez que se ejecuta DJNZ (código 1ØH = 16 decimal), se decrementa su contenido y se comprueba el nuevo valor mediante los indicadores del registro de estado, para ver si se ha hecho nulo. En caso de que no sea así, es decir que B no contenga cero todavía, tiene lugar un salto hacia detrás para continuar con una nueva iteración del bucle; para esto último es necesario especificar un octeto de desplazamiento como en la instrucción JR. Cuando el registro B pase a contener un valor de cero, el control pasará a la instrucción siguiente a DJNZ.

En la Figura 6.11, presentamos un programa demostrativo de esta instrucción, cuya única acción consiste en cargar B con 255, el máximo valor que puede tener un octeto (DJNZ trata el número almacenado en B como número sin signo). El octeto que provoca el salto hacia atrás es −2 (254 decimal), de modo que dicho salto es a la dirección del propio comando DJNZ otra vez. El programa completo le permite ver la rapidez de la cuenta, en relación con un bucle FOR...NEXT que usa los mismos números. En realidad, el código máquina termina mucho antes de lo que parece, debido al tiempo necesario para

que la parte BASIC responda en las líneas 1ØØ y 11Ø (lo detectará porque se escribe un Ø en pantalla, pues éste será el contenido final del par BC).

```
        LD B, 255
Atrás:  DJNZ Atrás
        LD C, B
        RET
```

```
 1Ø CLEAR 325ØØ: LET j=325ØØ
 2Ø FOR n=Ø TO 5: READ b
 3Ø POKE j+n, b: NEXT n
 5Ø DATA 6, 255, 16, 254, 72, 2Ø1
1ØØ INPUT "Pulse cualquier tecla..."; a$
11Ø PRINT USR j: PRINT "Terminado"
12Ø PRINT "Ahora pruebe la cuenta en BASIC"
13Ø INPUT "Pulse cualquier tecla..."; a$
14Ø FOR q=255 TO Ø STEP -1: NEXT q
15Ø PRINT "Terminado"
```

Fig. 6.11. Programa de «cuenta atrás» en código máquina, usando DJNZ y la correspondiente versión BASIC para comparar velocidades. ¡El resultado es engañoso!

La cuenta en código máquina en el último ejemplo es tan rápida que no podemos medir su duración, y lo que tarda sólo se debe a la parte BASIC que saca los resultados. Sería, pues, interesante, comparar una cuenta mucho más grande. DJNZ se puede usar únicamente para cuentas con un octeto; no se ha previsto una instrucción análoga para que un par de registros actúe de contador, así es que ésta es una buena oportunidad para conocer el método que permite cuentas con valor inicial de más de un octeto. Empezamos por cargar el par BC de registros, en este caso, con el mayor número que admiten dos octetos, FFFFH, que es 65535 decimal. La cuenta primero decrementa BC, pero como esa operación sobre registros dobles no altera los indicadores de estado en el Z-8Ø, hemos de detectar el fin de la cuenta de algún otro modo. La cuenta terminará cuando los dos registros del par BC sean cero, y el procedimiento usual de comprobar eso es cargar uno de los registros en A, y efectuar un "o-inclusivo" (OR) del acumulador y el otro registro. Si cualquiera de los operandos tiene un bit con valor "1" en cualquier posición, esta operación reproduce dicho bit en el mismo lugar del resultado, y el indicador de cero tendrá valor "Ø". Como resultado, el programa salta hacia atrás e inicia una nueva iteración del bucle al ejecutar la instrucción JR NZ (el salto es de tres

posiciones hacia atrás, o sea, −5 decimal, que corresponde al octeto de desplazamiento 251). Tanto el programa como los números decimales de código máquina son los de la Figura 6.12.

```
          LD BC, FFFFH1, 255, 255
Cuenta:   DEC BC          11
          LD A, B         12Ø
          OR C            177
          JR NZ, Cuenta   32, 251
          RET             2Ø1
```

Fig. 6.12. Versión en ensamblador de una «cuenta atrás», con un valor inicial mucho mayor almacenado en BC. Véase el método de comprobar el fin del bucle.

El programa de la Figura 6.13 utiliza esta nueva cuenta para ilustrar la diferencia de velocidades mejor que antes. Encontrará que la cuenta en código máquina tarda alrededor de medio segundo. En cambio, la misma cuenta programada en BASIC dura varios minutos. Eso le dará una idea de la rapidez que puede obtener con el código máquina. Aparte de todo, los bucles de ese tipo se emplean en temporizaciones. Como el cristal de cuarzo del Spectrum oscila a una veloci-

```
 1Ø CLEAR 325ØØ: LET j=325ØØ
 2Ø FOR n=Ø TO 8: READ b
 3Ø POKE j+n, b: NEXT n
 5Ø DATA 1, 255, 255, 11, 12Ø, 177, 32, 251, 2Ø1
1ØØ INPUT "Pulse cualquier tecla...";a$
11Ø PRINT USR j: PRINT "Terminado"
12Ø PRINT "Ahora pruebe la cuenta en BASIC"
13Ø INPUT "Pulse cualquier tecla...";a$
14Ø FOR q=65535 TO Ø STEP -1: NEXT q
15Ø PRINT "Al fin termine !!"
```

Fig. 6.13. Programa que le permite comparar la velocidad de ejecución del código máquina y de BASIC.

dad de 3.5 millones de pulsaciones por segundo (3.5 MHzs de frecuencia), contando el número de pulsos que necesita un programa, es posible calcular la duración de su ejecución. En el Apéndice E se incluyen los tiempos de las instrucciones más importantes, y en la Figura 6.14 puede observar la aplicación a nuestro ejemplo. La suma de los pulsos que hay en el bucle entre DEC BC y JR NZ resulta ser de 26 pulsos; por ello, para 65535 iteraciones se precisan más o menos

1,7Ø3,91Ø pulsos. Añadimos, si queremos, el tiempo que duran las instrucciones LD BC y RET para tener un total más exacto; estrictamente hablando, deberíamos contar el tiempo de JR NZ en 65534

Operación	Tiempo	Comentario
DEC BC	6	La misma duración que INC
LD A, B	4	
OR C	4	La misma duración que ADD C
JR NZ	12	En cada iteración
TOTAL	26	ciclos de reloj en el bucle

Fig, 6.14. Suma de los tiempos de ejecución de las instrucciones del bucle.

"pasadas", y usar el tiempo más corto en la última ejecución, en la cual no se salta ya. No obstante, sería una corrección despreciable frente al tiempo total. Según la estimación anterior, el bucle durará (1703910)/(3500000) segundos, aproximadamente Ø,48 segundos, lo que parece estar de acuerdo con la práctica.

Mediante estos bucles se pueden producir retardos de tiempo que, puesto que la velocidad del reloj está controlada por un cristal de cuarzo (como en los relojes digitales), serán muy precisos. Estos retardos de tiempo se usan numerosas veces en la ROM del Spectrum; las rutinas de impresión de la pantalla de T.V., del comando BEEP, de control de entradas-salidas del magnetofón y de la impresora, son los ejemplos más obvios.

Capítulo 7

Entradas y salidas

Grabación y lectura del código

Hasta este momento, hemos creado programas en código máquina con rutinas BASIC que introducían éste en la memoria. Esto es fácil de hacer, y no hay motivo por el cual no pueda grabarse todo el programa como sabemos hacer con BASIC. Si utiliza el desensamblador Campbell, verá que permite colocar el código en memoria, utilizando números hexadecimales en vez de decimales, y esto puede ser más rápido que escribir bucles READ...DATA en BASIC. Ahora tiene el problema de cómo grabar estos programas en la cinta, cuando no existe un programa BASIC que lo haya generado. El mismo problema surge cuando se emplea el ensamblador ULTRAVIOLET de ASC para producir código máquina que debe ser grabado y relocalizado. Incluso en el caso de tener una rutina BASIC que escribe el programa en memoria, puede que quiera grabar éste separadamente, para cargarlo de nuevo en el Spectrum, sin cargar, a la vez, la rutina BASIC. Afortunadamente, el Spectrum, a diferencia de sus predecesores, tiene previstos la grabación y carga de código máquina directamente, aunque, como usted habrá supuesto, ha de ser mucho más específico acerca de lo que desea hacer. La sintaxis se resume en el Capítulo 30 de manual del Spectrum; pero algunas prácticas con nuestros programas le darán mayor confianza en el manejo de este grupo de comandos.

Tenemos el programa con el que finalizó el Capítulo 6, que ocupa 9 octetos, empezando en la dirección 325ØØ. La sintaxis de un comando SAVE para este caso será:

SAVE "Cuenta" CODE 325ØØ,9

Por supuesto, puede elegir su propio nombre para el programa. Cuando presione ENTER aparecerá el mensaje habitual de arrancar el cassette y pulsar una tecla cualquiera. El programa se graba de forma muy parecida a como lo hacen los programas BASIC, por lo que debe tomar nota de dónde está en la cinta, para después poder rebobinar y cargarlo de nuevo.

Para hacer una prueba verdadera, desconecte el computador (tal vez prefiera salvar primero el programa BASIC que lo carga, ¡por si acaso!) para borrar por completo la memoria cuando conecte de nuevo. Como al apagar y encender el ordenador restablece las condiciones iniciales, tendrá que teclear CLEAR 325ØØ para reservar espacio a su programa. Una vez que lo haya hecho, escriba:

LOAD "Cuenta" CODE 325ØØ,9

y pulse ENTER, tras poner en marcha el magnetofón. Después de esas operaciones, el programa se cargará en el computador, comenzando en la posición 325ØØ. Verá el mensaje:

Bytes: Cuenta

que le recuerda que lo que se está leyendo es un programa en código máquina llamado Cuenta. Cuando se haya completado la lectura del programa, aparecerá el mensaje usual Ø OK, Ø:1 al pie de la pantalla.

Existen una serie de variaciones del LOAD de este conjunto de comandos. La versión que hemos citado arriba es la más completa, y obliga a que el computador compruebe cuidadosamente los errores posibles, de tal manera que si hay más octetos grabados en la cinta de los que señaló usted en el parámetro de longitud (el último), se obtendrá un informe de error. Puede usar también:

LOAD "Cuenta" CODE 325ØØ

cuando no recuerde la longitud del programa o no la sepa. Esto empezará la carga a partir de la dirección 325ØØ, y se almacenarán tantos octetos como contenga la cinta. Si no hay espacio en la memoria, tendrá que intentar la carga en direcciones diferentes; sin embargo, no todos los programas funcionarán correctamente al cambiarlos de posición en la memoria, aunque nuestro ejemplo sí lo hará. Por tanto, con el programa aún en la dirección 325ØØ, pruebe a recargarlo así:

LOAD "Cuenta" CODE 325Ø6

Ese comando llevará a cabo una nueva operación de lectura, empezando esta vez a almacenar octetos a partir de 325Ø6, y borrando, entonces, cualquier cosa existente en esas posiciones con anterioridad (en particular quedará sobre una parte del programa que estaba en 325ØØ). Para cerciorarse, ejecute el programa mediante:

PRINT USR 325Ø6

que, tras una breve pausa, retornará a BASIC con un resultado de cero.

Lo que no debe hacer ahora, bajo ninguna circunstancia, es poner PRINT USR 32500 para ejecutarlo de nuevo. En esta ocasión no ocurrirá nada, dada la manera de actuar del programa, salvo que saldrá de la primera copia y comenzará a ejecutar la nueva (no olvide que parte de ambas está superpuesta). Pero, generalmente, el resultado será un bloqueo del computador, que dejará de responder al teclado y le obligará a desconectar y volver a conectar. Si ignora la dirección del principio del programa y la longitud del mismo, el Spectrum aún se hará cargo de la situación. Para ello, utilice la variante:

LOAD "Cuenta" CODE

sin ningún parámetro, el Spectrum cargará los octetos en la posición de memoria que ocupaban cuando fueron grabados, que en nuestro programa era desde 32500 hacia delante. Este es un método siempre seguro, supuesto que no haya reservado suficiente espacio en la memoria (con CLEAR).

Retornando a la idea de localizar el código máquina en una parte de la memoria distinta a la ocupada originalmente, eso será posible únicamente si el código máquina es *independiente de la posición*. Se entiende por código independiente de la posición, aquel que no utiliza direcciones completas dentro de las instrucciones que se refieran a las posiciones en que estaba almacenado, o a direcciones donde va a ser colocado ahora. Por ejemplo, suponga que tiene un programa cuyo principio está en 32500, y que termina en 32599. Si en él existe una instrucción como JP 32510 o CALL 32577, esas direcciones estarán escritas en el interior del código del programa. En caso de intentar desplazarlo a las direcciones entre 32000 y 32099 (admitiendo que había suficiente espacio reservado), descubriría que el programa ya no funciona bien y tal vez ocurriese un bloqueo. ¿Por qué? Porque todavía tiene las instrucciones JP 32510 o CALL 32577, que se refieren a posiciones en las que ahora puede haber otros códigos completamente diferentes, o incluso ceros solamente. Este tipo de programas no se puede relocalizar de forma elemental; y un razonamiento análogo se hubiera aplicado si hubiese habido una llamada a la dirección 32000 en el programa, ya que al trasladar éste a ese rango de posiciones, lo que hubiese almacenado antes allí se ha borrado, es decir, el código al que se refería la llamada ya no existe. Para cambiar de sitio en la memoria programas de esta clase, hay que cambiar las direcciones que contienen las instrucciones (vea el Capítulo 8 para relocalizar programas creados con el ensamblador ULTRAVIOLET). En cambio, cuando su programa posee todos los saltos en forma relativa (JR) y lo mismo sucede con todas las llamadas a subrutina, o bien estas últimas se refieren al

sistema operativo que, por estar en ROM, no varía nunca y no puede borrarse, sería posible llevarlo a cualquier parte de la memoria RAM y ejecutarlo. Esta clase de código es "relocalizable", y el comando LOAD "Nombre" CODE dirección de comienzo, le permite almacenarlo para ello.

Dispone además de una rutina VERIFY para el código máquina, que sirve para asegurarse de que un valioso programa se ha grabado correctamente. Esta es especialmente útil con los programas cuyo código no se genera desde un programa BASIC empleando rutinas con READ...DATA y POKE.

Control de un puerto E/S

Como vimos en el Capítulo I, un puerto E/S (de entrada-salida) es un circuito que permite el intercambio de señales del sistema microprocesador (UCP y memorias RAM y ROM) con el exterior. En lo que respecta a la UCP, un puerto es otra dirección que puede utilizarse de forma muy parecida a la memoria, pero con un juego de instrucciones más reducido. Las más simples de las instrucciones para los puertos, en el Z-80, son IN A, (puerto), A (en ensamblador), y el BASIC del Spectrum proporciona comandos que efectúan directamente operaciones con los puertos desde el propio BASIC: IN y OUT. Como ya supondrá, la versión en código máquina es mucho más rápida y requiere especificaciones más detalladas.

Antes de empezar a ver cómo se emplean las instrucciones de código máquina para los puertos, es útil estudiar cómo funcionan los comandos BASIC equivalentes. Los comandos IN y OUT actúan igual que PEEK y POKE respectivamente, y deben ir seguidos de una dirección que no tiene nada que ver con las direcciones de memoria. Hay, por ejemplo, ocho direcciones que conectan los pulsadores del teclado con el microprocesador. Estas direcciones, cada una de las cuales se ocupa de la mitad de una fila de teclas, están detalladas en la Figura 7.1. También se citan en el Capítulo 23 del manual del Spectrum. Supongamos que escogemos una para probar. La Figura 7.2 presenta una rutina sencilla que lee el puerto 65Ø22 (una dirección inexistente en la memoria del Spectrum de 16k). Este puerto maneja las teclas de A á G en la segunda fila del teclado, y si ninguna de ellas se presiona, entonces el resultado es 255 cuando se ejecuta el comando IN 65Ø22. Es necesario incluir un bucle como para INKEY$, escribiendo:

```
2Ø LET x = IN 65Ø22: IF x = 255 THEN GOTO 2Ø
```

Dirección de Puerto	Teclas (de izda. a drcha.)
65278	De CAPS SHIFT a V
65Ø22	De A a G
6451Ø	De Q a T
63486	De 1 a 5
61438	De Ø a 6 (desde ahora el orden es de drcha. a izda.)
57342	De P a Y (¡equivocada en el manual!)
4915Ø	De ENTER a H
32766	De SPACE a B

Fig. 7.1. Direcciones de puerto para el teclado. Observe que cada puerto maneja la mitad de una fila de teclas.

```
1Ø PRINT "Pulse A ó G"
2Ø LET x=IN 65Ø22: IF x=255 THEN GOTO 2Ø
3Ø IF x=254 THEN PRINT "Esa era la A": GOTO 9999
4Ø IF x=239 THEN PRINT "Esa era la G": GOTO 9999
5Ø PRINT "Ha hecho trampa"
```

Fig, 7.2. Rutina BASIC que maneja el puerto del teclado.

Los números que se leen según las teclas pulsadas, aparecen en la lista de la Figura 7.3: observe que se leerán los mismos números, aunque se haya pulsado cualquiera de las teclas SHIFT al mismo tiempo. Volveremos más tarde sobre ese punto; por el momento,

CODIGO (decimal, hexa y binario)	Posición de la tecla en la mitad de la fila
254 FE 11111111Ø	Primera
253 FD 1111111Ø1	Segunda
251 FB 11111Ø11	Tercera
247 F7 1111Ø111	Cuarta
239 EF 111Ø1111	Quinta

Fig. 7.3. Códigos leídos según la tecla pulsada en cada puerto.

podemos utilizar el puerto en un programa de pulsación de tecla única como el de la Figura 7.2. Esta forma de investigar qué tecla se ha apretado, es mucho más rápida que las líneas:

```
1ØØ LET a$=INKEY$: IF a$="" OR CODE a$=13 THEN
  GOTO 1ØØ
2ØØ IF a$="g" THEN PRINT "Esa era la G"
```

y alguna más que habría que añadir. Es factible ir más allá, e inspeccionar dos puertos para detectar dos teclas pulsadas, que es como el propio Spectrum se organiza para generar códigos distintos, según estén pulsadas o no las teclas de "SHIFT" a la vez que las de las letras. Las teclas a las que se accede desde diferentes puertos, generan la misma secuencia de números, 254, 253, 251, 247 y 239. Así pues, podemos escribir un programa como el de la Figura 7.4 para comprobar si se están apretando dos teclas. Es muy elemental y sería posible mejorarlo bastante, aunque no importa, ya que hace lo que queríamos.

```
1Ø PRINT "Pulse G y T"
2Ø LET x=IN 65Ø22: LET y=IN 6451Ø
3Ø IF x=255 OR y=255 THEN GOTO 2Ø
4Ø If x=239 AND y=239 THEN PRINT "Muy bien":
 GOTO 9999
5Ø PRINT "!TRAMPA!"
```

Fig. 7.4. Programa en BASIC para detectar dos teclas pulsadas a la vez.

Pensemos en cómo lograr lo mismo con lenguaje ensamblador. Primero tendremos que escribir un organigrama, y en la Figura 7.5 aparecerá una sugerencia. Nos enfrentamos con el problema de leer dos puertos, y sólo cuando ambos informen de una tecla pulsada, seguiremos adelante (eso ocurrirá cuando ninguno de los puertos devuelva un 255). Según el organigrama, se comprueba el primer puerto, y si el octeto de entrada es 255, el programa vuelve a un bucle para repetir la comprobación del mismo puerto. Si el octeto devuelto desde el puerto es menor que 255, lo que indica la tecla oprimida en la correspondiente mitad de la fila controlada por él, se pasará a investigar el otro puerto. En caso de que la lectura de éste sea 255, o sea, no se ha pulsado la segunda tecla, el programa regresa al principio de todo para volver a repetir el proceso. Es necesario hacerlo así, pues si sólo volviese a examinar el segundo puerto, estaríamos detectando la presión sobre dos teclas consecutivas, no simultáneas; pero aún eso podría ser útil (por ejemplo, podría construir un programa que hiciese algo tecleando SI, y otra cosa tecleando NO). En el instante en que se aprieten ambas teclas en sus correspondientes medias filas, el resultado se lleva al par BC de registros (un octeto a cada uno), de forma que después se devuelvan al programa BASIC.

Parece todo muy razonable; no obstante, los comandos "IN A, (puerto)" del lenguaje ensamblador exigen una dirección de puerto de

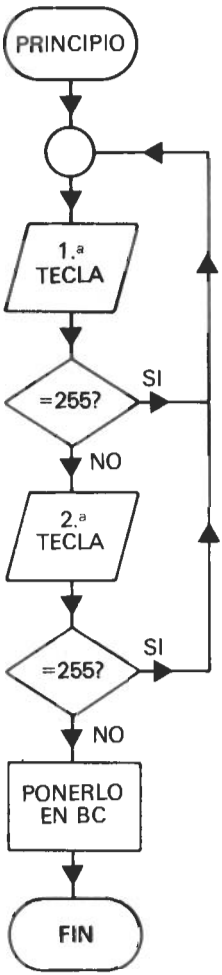


Fig. 7.5. Organigrama para una versión del programa de dos teclas en lenguaje ensamblador.

un único octeto, mientras que las direcciones del manual son de dos. Si intentamos cargar el acumulador con el contenido de dichas direcciones, en vez de cargar desde un puerto, encontraremos que no existen en un Spectrum de 16k (no hay memoria allí). La clave reside en el tipo de instrucción de entrada desde puertos que es necesario emplear. El Spectrum utiliza una clase de comando diferente en sus rutinas de lectura del teclado, que permite direcciones de puerto de dos octetos. Si se escribe la instrucción "IN A, (C)", el número almacenado en el par (BC) se toma como una dirección, usándose la parte baja contenida en el registro C como la dirección del puerto. Al mismo tiempo, la parte

superior de dirección del registro B aparece en las líneas de direcciones, y, de ese modo, el Spectrum direcciona el teclado. Como ya habrá imaginado, esto mismo se hacía en el ZX-81.

Para seguir los pasos del organigrama, entonces, tenemos que colocar las direcciones apropiadas en el par BC, y éstas son precisamente las que ya conocemos por el manual; aunque, eso sí, será necesario dividirlas en dos octetos de la forma habitual para cargarlas en BC. La versión en ensamblador del programa de dos teclas se muestra en la Figura 7.6, y para mayor claridad del ejemplo se comprueban los

```
Princ.:   LD BC, 65Ø22      1,254,253
          IN A, (C)          237,12Ø
          CP 255             254,255
          JR Z, Princ        4Ø,247
          LD D, A            87
          LD B, 251          6,251
          IN A, (C)          237,12Ø
          CP 255             254,255
          JR Z, Princ        4Ø,239
          LD C, A            79
          LD B, D            66
          RET                2Ø1
```

Fig. 7.6. Programa en ensamblador que detecta la pulsación de dos teclas a la vez, empleando la instrucción IN A, (C).

mismos puertos que en la versión BASIC. El grupo de teclas QWERT corresponde a los dos octetos de dirección 254, 253 (menor y mayor peso respectivamente), y el grupo ASDFG a 254, 251; por lo tanto, inicialmente cargamos BC con 253, 254 (en ese orden) y luego únicamente habrá que cambiar el octeto de B para examinar el otro puerto. Fíjese en cómo salvamos el primer valor encontrado en el registro D, porque A se utiliza para almacenar los octetos leídos en los puertos en ambas ocasiones. Al final del programa, los valores encontrados se copian en BC para ser examinados y devueltos a BASIC. La lista de números leídos en los puertos se da en la Figura 7.7, y se observa en ella, que son los mismos que leía la versión BASIC del programa.

Se presenta primero el ejemplo de g y t, porque es el que realizamos antes.

Experimentaremos ahora con un puerto de salida. El puerto 254 emplea tres bits para controlar el color del borde de pantalla, uno para

```
10 CLEAR 32500: LET j= 32500
20 FOR n=0 TO 20: READ b
30 POKE j+n, b: NEXT n
40 PRINT USR j
100 DATA 1, 254, 253, 237, 120, 254, 255, 40,
    247, 87, 6, 251, 237, 120, 254, 255, 40, 239,
    79, 66, 201
```

Teclas pulsadas	Número	Análisis (B, C)
g y t	61423	239,239
a y q	65278	254,254
w y s	65021	253,253
e y d	64507	251,251
f y r	63479	247,247

Se presenta primero el ejemplo de g y t porque es el que realizamos antes.

Fig. 7.7. Programa BASIC para almacenar el código máquina y los resultados de pulsar las teclas.

la toma MIC y otro para el altavoz. El octeto se divide como indica la Figura 7.8, con los tres bits menos significativos (llamados D0, D1 y D2) para formar el número de color del borde, D3 para enviar un bit al conector MIC, y D4 para enviar otro bit al altavoz.

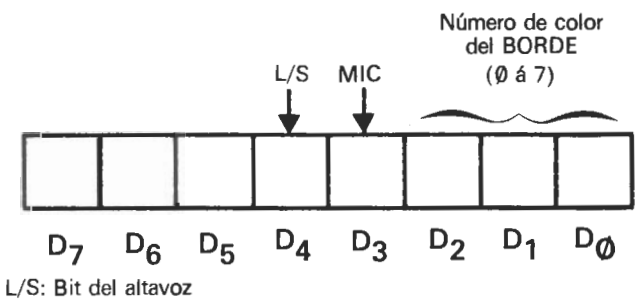


Fig. 7.8. Significado de los bits del puerto 254.

Podemos probar un pequeño programa en ensamblador que envíe algo al bit del altavoz de este puerto y ver qué ocurre. Partimos del organigrama de la Figura 7.9 y obtenemos el programa ensamblador de la Figura 7.10. Cuando ponemos éste en la memoria (Figura 7.11) y lo ejecutamos, lo más obvio es que el borde cambia a color negro. Una pequeña reflexión aclara este comportamiento, debido a que estamos utilizando los números 0 y 16 para enviarlos al puerto desde el programa. El 0 decimal es 00000000 en binario, y 16 decimal es el 00010000

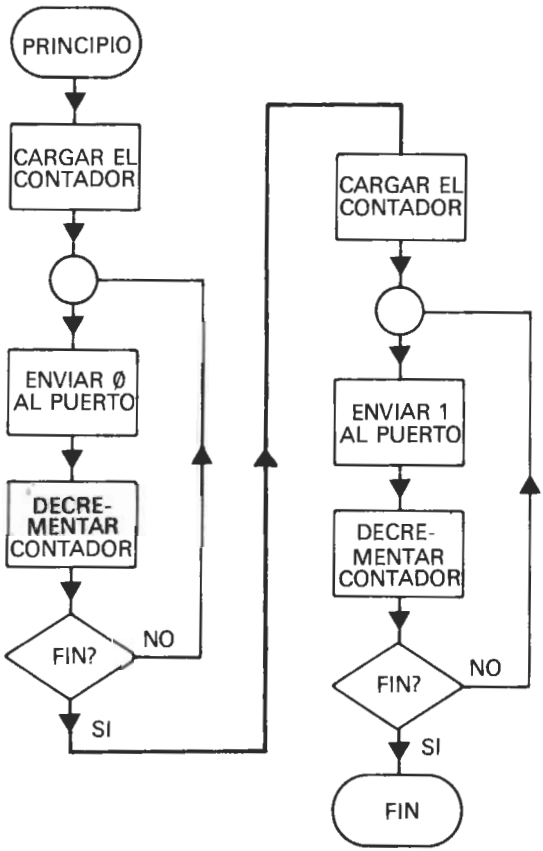


Fig. 7.9. Un organigrama para el envío de bits al puerto del altavoz.

```
        LD B,255              6,255
        LD A, Ø               62,Ø
Sal 1:  OUT (puerto), A       211,254
        DJNZ, Sal 1           16,252
        LD B, 255             6,255
        LD A, 16              62,16
Sal 2:  OUT (puerto), A       211,254
        DJNZ, Sal 2           16,252
        RET                   2Ø1
```

Fig. 7.10. El programa en lenguaje ensamblador.

binario. En cada caso, los tres bits de menor peso son cero, que corresponden al número de color Ø, y ése es el color que adquiere el borde: negro. Suponga que colocamos en esos bits el número 4 decimal, que significaría poner 1ØØ en ellos. Eso se consigue cargando el

```
10 CLEAR 32500: LET j = 32500
20 FOR n = 0 TO 16: READ b
30 POKE j + n, b: NEXT n
50 DATA 6, 255, 62, 0, 211, 254, 16, 252, 6,
   255, 62, 16, 211, 254, 16, 252, 201
100 PRINT USR j
```

Fig. 7.11. Programa cargador del código máquina.

acumulador con 4 y 16 + 4 = 20, en lugar de 0 y 16; así pues, cambiamos apropiadamente los octetos de datos. Efectivamente, aparece el borde de color verde; luego, ¡hemos descubierto cómo controlar el color del borde desde código máquina! A pesar de todo, eso no era lo que queríamos hacer, y si usted escucha muy atentamente al ejecutar el programa, notará que, al pulsar ENTER, se oye un "click" doble. Uno de ellos es el que se produce normalmente al pulsar una tecla, pero el otro es el resultado del programa. Este ha enviado un 0 y después un 1 al altavoz, provocando un "click" adicional.

El ejercicio siguiente consiste en prolongar esto algo más para escucharlo con claridad. Para ello, tenemos que repetir el "click" varias veces y suficientemente rápido, de modo que sea más apreciable, y eso, a su vez, significa otro bucle. De nuevo, hemos de dibujar un organigrama que nos indique la secuencia de pasos que necesitamos dar, y la Figura 7.12 lo ilustrará. Es muy breve, pues la rutina completa del "click" que vimos antes en la Figura 7.9 se ha expresado como un único bloque de acción, en vez de desarrollarse en detalle. Lo siguiente es pasar de este diagrama a un programa en ensamblador.

Existen varias formas de hacerlo, y es instructivo comentar más de una. Dado que se precisa otra cuenta, sería muy útil poder usar DJNZ otra vez. Sin embargo, eso exige tener cierto cuidado, puesto que hemos incluido dos veces esa instrucción en la rutina del "click", y el registro B contiene un cero al terminar dicha rutina. Si ponemos un valor en B para que sirva de contador, será alterado por las instrucciones DJNZ de la rutina que provoca el "click". No obstante, es factible cargar B con un número, meterlo en la pila mediante PUSH BC, ejecutar la rutina del "click", extraer B de la pila con POP BC, ejecutar luego DJNZ y, si al decrementarse todavía no es cero el contenido de B, saltar hacia atrás para repetir el bucle, empezando por salvar otra vez BC en la pila. Recuerde que la pila es una porción de memoria RAM que constituye un almacén temporal, donde el microprocesador pone a salvo el contenido de los registros, para evitar que se borren y poder utilizarlos más tarde. Si efectuamos un PUSH BC antes de la

rutina del "click", el número que estaba en B volverá a éste cuando se ejecute la instrucción POP BC con su valor inalterado.

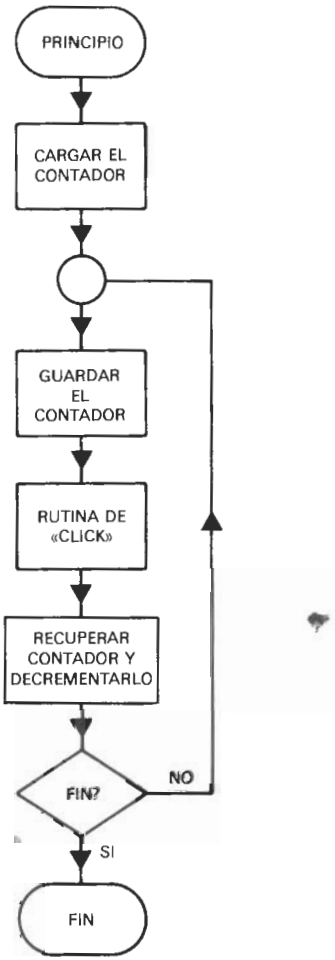


Fig. 7.12. Organigrama del programa «zumbador».

Otra alternativa es abandonar la idea de incluir un nuevo DJNZ, y utilizar un registro (o un par de ellos) diferente como contador adicional. Se carga un registro simple con un valor, y se decrementa al final de cada "click"; después se comprueba el resultado para ver si es cero, y, si no lo es, volver a repetir todo el proceso. Cuando el nuevo contador sea nulo, terminará el programa. En caso de escoger un par de registros como contador, tenga presente que las instrucciones de decremento sobre ellos no alteran los indicadores del registro de estado. Sea cual sea el método elegido para contar el número de "click" que

vamos a producir, la rutina que causa los mismos podría ponerse en forma de subprograma y llamarse cada vez que sea necesario.

Consideraremos el método de la instrucción DJNZ, y en la Figura 7.13 mostramos los nuevos comandos añadidos. El programa requiere unas pequeñas alteraciones respecto del programa que producía el "click" aislado, y aprovecharemos para cambiar los valores enviados al puerto por 7 y 23, con lo que el borde permanecerá blanco.

```
        LD B, 255
Atrás:  PUSH BC
        ... Rutina de "click"...
        POP BC
        DJNZ, Atrás
        RET
```

Fig. 7.13. Cómo emplear DJNZ para mantener dos cuentas diferentes. El contenido del par BC se guarda en la pila temporalmente.

El resultado se ve en la Figura 7.14. Produce un "zumbido" que verifica la corrección del procedimiento elegido, y ahora intentaremos alterar algunos valores. Por ejemplo, si cambiamos el octeto 255 que aparece en la rutina de "click" por un 128, lo ejecutamos y después probamos con 5Ø volviendo a ejecutarlo, veremos (o mejor dicho oiremos) cómo varía la nota según el retardo de tiempo: los retardos más cortos dan notas de tono más alto.

```
 1Ø CLEAR 325ØØ: LET j=325ØØ
 2Ø FOR n=Ø TO 22: READ b
 3Ø POKE j+n, b: NEXT n
 5Ø DATA 6, 255, 197, 6, 255, 62, 7, 211, 254,
    16, 252, 6, 255, 62, 23, 211, 254, 16,
    252, 193, 16, 236, 2Ø1
1ØØ PRINT USR j
```

Fig. 7.14. Programa que produce un «zumbido», con su cargador BASIC.

Para confirmar que deben enviarse ambos bits 1 y Ø, al altavoz, haremos que el programa envíe un solo bit. Observará que el borde cambia de color, pero no se escucha ningún sonido (ver Figura 7.15). Recuerde que el desplazamiento para la orden DJNZ del bucle externo tiene que cambiarse, cada vez que varíe el número de octetos que componen el programa. Los bucles pequeños incluidos en la rutina del "click" permanecen como estaban.

```
10 CLEAR 32500: LET j=32500
20 FOR n=0 TO 14: READ b
30 POKE j+n, b: NEXT n
50 DATA 6, 255, 197, 6, 255, 62, 0, 211,
   254, 16, 252, 193, 16, 244, 201
100 PRINT USR j
```

Fig. 7.15. Comprobación de que no hay «zumbido» si se envía un sólo bit al altavoz.

Es posible crear algunos efectos visuales interesantes, variando los octetos de color del borde, cada vez que se ejecuta la rutina que produce el sonido "click". La forma más sencilla de hacerlo es cargar en el acumulador el contenido del registro B que sirve de contador global, y enviarlo al puerto. Puesto que el número cambia en cada iteración del bucle exterior, también cambiará el color del borde. La Figura 7.16 presenta alguna sugerencia al respecto.

```
        LD B, 255          6,255
bucle:  PUSH BC            197
        LD A, B            120
salida: OUT (puerto), A    211,250
        JNNZ, salida
        POP BC             193
        DJNZ, bucle        16,247
        RET                201

10 CLEAR 32250: LET j=32500
20 FOR n=0 TO 11: READ b
30 POKE j+n, b: NEXT n
50 DATA 6, 255, 197, 120, 211, 250, 16, 252,
   193, 16, 247, 201
100 PRINT USR j
```

Fig. 7.16. Envío de números al puerto. Observe los efectos de color y sonido, pero ¡desconecte su impresora!

Con el mismo puerto controla, además, el conector MIC del Spectrum, que podemos usarlo para enviar señales al magnetofón, y tal vez sería posible escribir una rutina que transmitiese señales en serie, de forma estándar, para las impresoras (no para la impresora ZX) u otros dispositivos en serie. Una transmisión en serie consiste en mandar los bits de uno en uno, y hay que tener algún método para distinguir un octeto del siguiente, por lo que existen sistemas normalizados de transmisiones de

esta clase. El método estándar más conocido se denomina RS232, y en él se transmiten secuencias de once bits, uno por cada octeto enviado. Una secuencia comprende un "bit de arranque", que es un 1, seguido de los ocho bits del octeto que se está mandando, y, por último, "dos bits de parada", ambos ceros (Figura 7.17). Los mensajes se transmiten en código ASCII, y el tiempo entre cada dos bits es constante, utilizándo-

Fig. 7.17. Uso de los bits de «arranque» y de «parada» para la transmisión en serie.

se para él diversos valores normalizados. Normalmente, no se suele dar ese tiempo entre bits, sino el número de bits enviados por segundo, o velocidad en Baudios (ésta no es una definición formal del Baudio, pero, por el momento, nos vale). Para los antiguos teletipos se usaban velocidades de 110 Baudios, que correspondían a 1Ø caracteres por segundo. Las modernas impresoras en serie aceptan señales que van desde 11Ø a 96ØØ Baudios (aunque no son capaces de imprimir a esas velocidades tan altas, únicamente aceptan señales en su memoria tampón). Las velocidades de transmisión en Baudios típicas se detallan en la Figura 7.18.

75 11Ø 15Ø 3ØØ 12ØØ
24ØØ 48ØØ 96ØØ 192ØØ

Fig. 7.18. Velocidades de transmisión normalizadas (en Baudios).

Escribir una rutina RS232 para el puerto del cassette no es exactamente el tipo de trabajo recomendado a un principiante; pero, con lo que ya sabe, puede entender cómo deberían trabajar las partes esenciales de una de tales rutinas. El bit de arranque se envía cargando un 15 en el acumulador, y mandándolo al puerto 254. Se toma 15, porque coloca el bit D3 a "1", y los bits DØ, D1 y D2 también, con lo que el color del borde no cambia. Es necesario programar un bucle de temporización para controlar el tiempo de duración de la señal enviada. Después, hay que llevar el octeto seleccionado al acumulador para transmitirlo. ¿Cómo se hace esto? Un método consiste en borrar el indicador de acarreo (la instrucción "OR A" lo hará), y luego se hace una rotación a la izquierda del acumulador, a través de dicho indicador. Tras cada rotación, se comprueba el bit de acarreo, y se llama a una rutina que envíe un 15 o un 7, según sea el resultado un "1" o un

"Ø", respectivamente. Entre cada dos de estas operaciones de salida al puerto, es preciso llamar a la rutina de retardo; una vez que se hayan transmitido los ocho bits (¡también habrá que contarlos!), se mandarán las dos señales de parada (dos bits de valor "1"), de nuevo con el mismo retardo entre ambos. Este es un esquema relativamente simple, e ignora un convenio llamado paridad, que sirve para descubrir errores de transmisión. La paridad consiste en enviar siempre un número par o impar de unos en cada octeto. Si se trabaja con paridad par, el número de unos de un octeto siempre deberá ser par, y esto es posible, porque RS232 se utiliza con los códigos ASCII que tienen sólo 7 bits. El octavo bit (el más significativo) se pone a "1" ó a "Ø", de manera que el octeto completo tenga un número par de unos. Lo mismo se aplica a la paridad impar. En el registro de estado hay un indicador de paridad, y gracias a él la programación de este método de detección de errores es muy simple. Para controlar una impresora en serie, la operación de paridad se ignora muy a menudo, y muchos circuitos permiten trabajar con señales de Ø y +5 voltios, en lugar de −12 y +12 voltios, que son las tensiones de trabajo del RS232 normalizado.

Esa es la parte fácil. La parte difícil es que el Spetrum almacena sus listados en forma comprimida, utilizando tokens en vez de palabras clave; en consecuencia, no se pueden enviar octetos leyéndolos directamente de la memoria. Aún así, es posible encontrar la rutina en ROM que el Spectrum posee para confeccionar sus listados de programas, e incluso se podría intentar interceptar las señales que el computador manda al puerto de la impresora, cuando lista los programas en ella. ¡Pero eso es ya un trabajo de expertos, no de novatos!

Capítulo 8

Depuración y más programación

Los encantos de la depuración

Ahora que conoce los placeres de la programación en código máquina, es el momento oportuno para hablar de la depuración. Los errores de los programas reciben el nombre pintoresco de "gazapos" (bug, en inglés), y la depuración es el proceso de eliminación de los mismos. Probablemente, existe un nombre para el programador que comete los errores, pero nuestra misión actual es descubrir la mejor forma de eliminarlos. La primera parte es la prevención. Repase cuidadosamente su organigrama, para asegurarse de que describe realmente lo que usted desea hacer, y luego compruebe exhaustivamente su programa en lenguaje ensamblador, para cerciorarse de que su resultado será el que se espera. Después, asegúrese de que los octetos que se almacenan en memoria corresponden a los códigos de operación de las instrucciones y a los datos del programa en ensamblador. Si lleva a cabo todo eso, eliminará un gran número de "gazapos", antes de que las cosas empiecen a "salirse de quicio". No piense que es usted torpe si no los detecta todos; salvo que el programa sea muy simple, hay una gran probabilidad de que existan errores en alguna parte, y eso nos ocurre a todos. Si utiliza un ensamblador, desaparece una de las mayores fuentes de errores casi instantáneamente. El proceso de traducir instrucciones de lenguaje ensamblador a octetos en la memoria buscando en unas tablas, es algo que propicia los fallos, y si se deja que el propio computador realice ese trabajo, se previenen un montón de errores potenciales. Describiré brevemente el ensamblador ULTRAVIOLET© más adelante, en este mismo Capítulo. En el momento de escribir este libro, el ULTRAVIOLET era el único ensamblador disponible para el Spectrum, aunque se estaba desarrollando otro basado en el uso del Microdrive. Es casi seguro que cuando aparezca el libro en el comercio, existirá ya toda una serie de ellos donde elegir. Si el código máquina le interesa realmente, y quiere usted dedicarse a trabajos mucho más avanzados que los que apa-

recen en este libro, un programa ensamblador será una herramienta indispensable.

Si, por el contrario, pretende servirse del código máquina sólo como un medio de efectuar tareas secundarias que son imposibles o demasiado lentas con BASIC, el método de introducir los octetos en memoria con comandos POKE, que ha aparecido a lo largo de nuestra exposición, resulta perfectamente adecuado. Sin embargo, eso significa que los "gazapos" estarán al acecho en cada rincón del código máquina. La causa principal de las equivocaciones es el aburrimiento. La conversión de programas en ensamblador a octetos en forma decimal es un trabajo tedioso, y todos los trabajos tediosos propician los errores. Una posibilidad es la conversión incorrecta de hexadecimal a decimal, y cambiar los números al escribirlos es otra que, sorprendentemente, ocurre a menudo. Los desplazamientos de las instrucciones JR y DJNZ constituyen una fuente de problemas potenciales. Se puede calcular un número de desplazamiento equivocado; o, tal vez, escribir la instrucción adecuada en un primer momento, añadir más tarde líneas al programa, y olvidarse de modificar los desplazamientos relativos. Ese tipo de fallos se soluciona igualmente cuando se dispone de un ensamblador. Un salto erróneo causará casi siempre un bloqueo del computador, o que éste entre en la rutina del comando NEW y, a menos que haya grabado previamente el programa fuente (es decir, el programa BASIC que carga el código en memoria o que tiene las instrucciones en ensamblador) o el propio código máquina, habrá perdido todo lo que hubiese dentro del ordenador, que muy bien podría ser el resultado de un gran esfuerzo. Otra forma de salto incorrecto, que es más complicado de descubrir, consiste en programar la condición opuesta a la deseada; por ejemplo, escribir JR Z en lugar de JR NZ. Se debe seguir con atención lo que haría el salto con diferentes octetos de datos, para eliminar este tipo de confusiones.

Un defecto muy común es utilizar los registros, suponiendo que inicialmente contienen cero. Nunca debe suponer eso: es mucho más seguro asumir que cada registro tiene un valor que estropeará todo el programa si no se remedia. ¿Qué hacer cuando se ha revisado todo el programa y, sin embargo, éste no funciona como se esperaba? No hay una respuesta única para esto. Es posible que su organigrama no haga lo que usted creía, y, si no dibuja uno nuevo, obtendrá lo que en realidad ha especificado involuntariamente. Tal vez esté llamando una rutina de la memoria ROM del Spectrum, que no opera del modo que esperaba; hasta que esté disponible el desensamblaje completo de la ROM, el único método es el de prueba y error. Lo más que puedo hacer, es darle algunos consejos acerca de cómo eliminar "gazapos" en

los programas que aparentemente están bien construidos, pero no trabajan como se pensaba.

La primera regla de oro es no probar nunca nada nuevo en medio de un programa largo. Su programa en código máquina, idealmente, estará compuesto de una serie de subrutinas, cada una de las cuales habrá comprobado concienzudamente antes de unirlas para constituir el programa completo. En la práctica, eso no es tan sencillo, porque el Spectrum no permite el comando MERGE (para unir programas), cuando se manejan rutinas en código máquina; no obstante, todavía es posible cargar el código grabado, en secciones de memoria que sean contiguas cada una a la anterior, o utilizar MERGE con las líneas de datos de los programas BASIC, y luego se introducen en memoria todos juntos, mediante POKE, que es un método más seguro que el anterior. Si almacena en cinta subrutinas bien depuradas, preferiblemente incluidas en un programa cargador en BASIC, después se evitará un montón de trabajo combinándolas, teniendo presente que si cualquiera de ellas no es relocalizable, será necesario alterar los octetos de direcciones que aparezcan en el programa. Una vez más, los usuarios de un ensamblador llevan la mejor parte, pues si las instrucciones de lenguaje ensamblador se almacenan dentro de sentencias REM, pueden ser unidas con MERGE y editadas antes de traducirse, felizmente para ellos.

Aun en el caso de que no construya una biblioteca de rutinas en cinta de cassette, ésta puede servirle para tener un índice de subrutinas. La revista "Personal Computer World" publica cada mes una serie denominada SUBSET (subconjunto), que trae varias rutinas en código máquina de propósito general, y la mayor parte de ellas son para el Z-8Ø, lo que refleja la importancia de este microprocesador. Aunque después no las use, la documentación que les acompaña debería darle algunas ideas sobre la forma de almacenar sus propias creaciones; personalmente, creo que esta serie justifica mi suscripción por sí sola. Cuando incluya una nueva subrutina en un programa, es bastante sensato ejecutarla primero independientemente, con el fin de asegurarse: (a) de todo lo que necesita tener en los registros antes de llamarla en el programa, y (b) de lo que contendrán los registros después de que la subrutina termine de ejecutarse. El proceso de planificación que hemos descrito debería eliminar los errores en su mayor parte. Si, a pesar de todo, se enfrenta con un programa que no desea examinar rutina a rutina, la mejor forma de tratarlo, supuesto que no dispone de un buen programa monitor, es insertar puntos de parada (break, en inglés). Un punto de parada, en lo que respecta a los programas en código máquina para el Spectrum, consiste en una instrucción RET (código 2Ø1). Cuando se encuentra ésta, el contenido del par BC se pasa al sistema operativo del Spectrum y se vuelve al funcionamiento

normal. Para descubrir errores en los programas por este camino, será necesario colocar un punto de parada en lugares del código donde pueda examinar valores y resultados al retornar el control a BASIC. Dichos valores podrían ser los contenidos de otros registros, por lo que antes de la instrucción RET, habrá que copiar el registro en cuestión en el par BC, para examinarlo más tarde. Los puntos de parada se incluyen en las líneas DATA del programa BASIC cargador. Al encontrarse uno de esos puntos durante la ejecución del código máquina, se retornará a BASIC con el valor almacenado en el par BC de registros, sobre el que usted puede empezar a discurrir. Si el programa funciona correctamente hasta un punto de parada, elimínelo y coloque otro en una etapa posterior. Repitiendo este proceso, encontrará la zona del programa que opera erróneamente (al menos teóricamente debería ser así) y a partir de ella puede detectar las instrucciones causantes del problema. Los fallos más graves son los bucles mal programados, ya que, invariablemente, conducen casi siempre al bloqueo, y en el Spectrum no hay manera de salvar esa situación. Algunas máquinas cuentan con una "inicialización" incluida en sus circuitos, o sea, un botón que, al pulsarse, restaura las condiciones normales de funcionamiento del ordenador, incluso cuando éste se ha bloqueado ejecutando un programa en código máquina erróneo. En el Spectrum normal no se dispone de este mecanismo de ayuda, aunque sí aparece en otros muchos sistemas basados en el Z-8Ø. Por lo tanto, es muy probable que alguno de los fabricantes independientes de accesorios para el Spectrum, ofrezca alguno que proporcione esa facilidad al computador y haga la vida más fácil a los programadores de lenguaje ensamblador.

Una equivocación que provoca frecuentemente bucles sin fin, es un salto hacia atrás a una dirección incorrecta. Por ejemplo, si tuviésemos un programa así:

```
        LD B, 255
Atras:  OUT (puerto), A
        DJNZ Atras
```

y lo tradujésemos "a mano", sería posible hacer que la instrucción DJNZ saltase a la dirección de "LD B, 255", en vez de a la etiqueta Atras. Eso provocaría que el registro B se cargase en cada iteración con el valor 255 y, como resultado, la instrucción DJNZ nunca conseguirá decrementar el contenido de B hasta cero, lo que significaría un bucle sin fin. Errores como éste se ponen de manifiesto rápidamente en lenguaje ensamblador, porque se puede comprobar a simple vista la situación de la etiqueta, pero resultan muy difíciles de encontrar cuando sólo se tienen octetos en código máquina. La única solución es, otra vez, comprobar cuidadosamente los bucles. En ese sentido, el método,

explicado en el libro, de escribir primero el programa en lenguaje ensamblador y traducirlo posteriormente a números, es una forma excelente de evitar el problema.

Monitores

He mencionado brevemente los monitores en el apartado anterior. La palabra "monitor" es desafortunada, porque con ella se designan dos conceptos diferentes asociados a los ordenadores. La definición de monitor en electrónica de equipos, se refiere a un tipo de pantalla de rayos catódicos que acepta señales de televisión directamente, sin que haya que transformarlas para su transmisión, que es el método que emplea el Spectrum con el televisor normal. Por otro lado, un monitor es un programa que comprueba (monitoriza) cada acción del computador, y ésta es la clase de monitor a la que me referiré en esta sección (a la pantalla de TV. la llamaré "monitor de vídeo").

Los monitores se han desarrollado a través de los años. En los primeros días de los computadores personales, lo más que se podía esperar de un monitor era que mostrase una sección de la memoria en hexadecimal, que cambiase el contenido de aquella de octeto en octeto, un desplazamiento de octetos de una dirección a otra, y la posibilidad de insertar puntos de parada. Algunos computadores que tenían monitores de código máquina incorporados, recibían el nombre de "paneles frontales", denominación confusa, muy en boga en la prehistoria de la informática. En la actualidad, los monitores disponibles consisten en programas (en cinta o disco), que amplían enormemente las posibilidades del computador para la depuración del código máquina. Ciertos monitores muy recientes, por ejemplo, permiten que un programa en código máquina (residente en memoria RAM o ROM, indistintamente) se ejecute paso a paso, es decir, instrucción por instrucción, y presentan, al completarse cada paso, el contenido de los registros y de la memoria (de las posiciones de memoria alteradas por esa instrucción). Un monitor de esa especie, de los cuales el STEP-8Ø de Mumford para TRS-8Ø es el ejemplo más conocido, es para un programador de código máquina lo que un taladro eléctrico para los amantes de las reparaciones domésticas; empieza uno a preguntarse cómo es posible vivir sin él.

Cuando estaba escribiendo este libro, nada más aparecer el Spectrum, no existía una gran variedad de monitores para el Spectrum. Yo he utilizado las ayudas de ese estilo que ofrece el Desensamblador de Campbell, que es uno de los programas de más utilidad que se han escrito para el Spectrum. Indudablemente, cuando se publique este libro se dispondrá ya de muchos más, por lo que la depuración del código máquina en el Spectrum resultará menos penosa.

Utilización de un ensamblador

En cualquier libro sobre lenguaje ensamblador y código máquina, es necesaria alguna descripción de cómo funcionan los programas ensambladores; omitirla sería como hablar del mantenimiento de un motor, sin mencionar las llaves fijas para las tuercas. Los ensambladores son tan útiles para la programación en código máquina, que no pasará mucho tiempo antes de que existan tantos de ellos para el Spectrum como para máquinas tan conocidas como el TRS-8Ø, a pesar de que ahora mismo sólo se dispone de uno.

Un ensamblador es un programa, generalmente en código máquina, que debe cargarse igual que cualquier otro. Esta es una operación bastante rápida, incluso con los magnetófonos. Una vez que el ensamblador se ha cargado, seguramente presentará una serie de opciones en la pantalla. Entre ellas figuran, típicamente, la introducción o edición de lenguaje ensamblador, la grabación o lectura de un programa en dicho lenguaje (denominado código fuente) o en código máquina (llamado código objeto), la traducción (o ensamblaje) del código fuente a código objeto, y el movimiento de un "puntero" del editor, que permite seleccionar la línea que se va a editar. Una sesión de escritura de código comenzaría con la selección de la opción "añadir código fuente".

Cada ensamblador goza de características propias, que frecuentemente reflejan las peculiaridades de la máquina en que se utiliza; pero la mayoría de los ensambladores para Z-8Ø siguen las normas establecidas por Zilog (la casa que diseñó el Z-8Ø). No obstante, en vez de tratar de los ensambladores en general, probablemente será más provechoso, en este momento, ilustrar este apartado tomando como referencia el primer ensamblador desarrollado para el Spectrum: el ULTRAVIOLET de ACS.

El ensamblador ULTRAVIOLET

Este traductor acepta los datos y direcciones en base decimal, y no es preciso, en la etapa de escritura, utilizar para nada el código hexadecimal. Esto permite introducir directamente los números decimales que aparecen en el manual del Spectrum. Todas las instrucciones del Z-8Ø se ensamblan correctamente; el código se puede colocar en memoria (con algunas precauciones) de manera distinta, y cambiarse de lugar si es necesario. Se pueden obtener listados completos del lenguaje ensamblador y del código máquina obtenido, tanto en pantalla como en la impresora ZX, que es muy útil cuando se desarrollan programas extensos.

Sin embargo, el estilo del ULTRAVIOLET es muy similar al de los ensambladores para el ZX-81, y no aprovecha al máximo la posibilidad que ofrece el Spectrum de reservar espacio para el código máquina en direcciones altas de la memoria. El programa requiere, también, que las sentencias de lenguaje ensamblador se escriban dentro de líneas REM de BASIC, como tenía que hacerse en el ZX-81, y el código se almacena, asimismo, en una línea REM: la primera del programa. A pesar de eso, el código ensamblado se puede grabar y, posteriormente, al leerlo, se coloca en posiciones altas de la memoria, con todas las direcciones del programa corregidas apropiadamente por el ensamblador. Aunque tiene la desventaja de ser una versión modificada de un programa para el ZX-81 (cosa inevitable, porque escribir de nuevo un ensamblador es una tarea compleja), constituye una herramienta de programación de gran valor para la creación de programas en código máquina.

ULTRAVIOLET, al igual que su correspondiente desensamblador INFRARED, está disponible en versiones de 16k ó de 48k, de las cuales yo he empleado la primera únicamente. Esta última consiste en una cinta de cassette que, una vez ajustado ligeramente el ángulo de la cabeza lectora de mi magnetofón, se cargó con facilidad. El sistema tiene una sección de carga, titulada "uv-loader", un programa principal en código máquina llamado "uv-16k", y una sección de pantalla que presenta unas cuantas instrucciones detalladas (como en una hoja de referencia impresa). Cuando se han leído las instrucciones, se pulsa cualquier tecla y aparece el mensaje de "copyright", como al conectar el ordenador. Este hecho, aunque alarmante, es perfectamente normal para este programa: significa que se ha cargado satisfactoriamente en la memoria y está listo para empezar.

Escritura de un programa

Es necesario escribir las instrucciones del lenguaje ensamblador con letras minúsculas y en una línea REM de BASIC. La primera línea REM debe contener suficientes espacios, o cualquier otro carácter, como para que quepa el código máquina, que se colocará en ella después de la traducción. Puesto que BASIC comienza en 23755, y los cuatro primeros octetos de la primera línea comprenden el número y la longitud de la misma, siendo el siguiente octeto el token de REM (el código numérico asignado a REM), los octetos del programa en código máquina se almacenarán a partir de la dirección 23760. La segunda línea del programa debe incluir la palabra "go" a continuación de REM (éste es el comando: "comienza la traducción"). En la tercera línea tiene que aparecer la dirección de origen, precedida de "org", que

normalmente será 23760. Cualquier intento de ensamblar el código en las direcciones altas de la memoria, tales como 32500, por ejemplo, probablemente borrará el propio programa ULTRAVIOLET y provocará un bloqueo del computador. Esto se puede evitar con una modificación de la sentencia "org", que especifica, entre paréntesis, una dirección de relocalización del código generado, después de la dirección de origen normal. Por ejemplo, la instrucción "org 23760 (32500)", hace que el código máquina se almacene a partir de 23760, pero de tal forma que se ejecutará correctamente cuando los octetos del programa se desplacen a la dirección 32500 (¡lo que significa que no funcionará bien en la posición 23760!). Para relocalizar el código, se graba éste y después se carga en la dirección apropiada. Pueden existir varias sentencias "org" para producir con el ensamblador diversas rutinas de código máquina, que se ejecutarán más tarde en diferentes direcciones.

Después de las sentencias citadas, se introduce ya el programa en lenguaje ensamblador, según las reglas del sistema ACS. Cada línea debe comenzar con un REM, seguido de las instrucciones de ensamblador en *letras minúsculas*, análogamente a como aparecen en el manual del Spectrum. Los nombres de etiquetas *tienen que empezar con una letra mayúscula* y pueden tener cualquier longitud. Es mejor, sin embargo, restringirse al uso de nombres cortos para ahorrar memoria, sobre todo en el Spectrum de 16k, al cual le queda una escasa cantidad libre cuando se ha cargado el ensamblador ULTRAVIOLET (cuyo código ocupa casi 5k por sí solo). Los paréntesis de cierre y el signo "más", son los únicos caracteres que no pueden formar parte de los nombres de etiqueta. El punto y coma sirve para separador entre la etiqueta y la sentencia de lenguaje ensamblador que viene detrás.

Los comentarios van precedidos por el signo de exclamación, en lugar de por el punto y coma estándar en el ensamblador del Z-80. En una misma línea pueden existir hasta 32 caracteres de un comentario; si se precisan más, hay que ponerlos en una nueva línea REM. Se admiten varias instrucciones detrás de un mismo REM (generalmente, salvo en programas largos, se podría escribir todo el programa en una sola línea), pero las diferentes sentencias deben separarse con punto y coma. Esto último es esencial, pues si se emplean los dos puntos como en BASIC, el sistema operativo del Spectrum tratará la siguiente tecla como una palabra clave, en lugar de como una letra.

El final del programa se señala con otra línea REM, que contendrá la palabra "finish". El programa BASIC resultante se comprueba y se edita en la forma habitual, y, cuando esté todo bien, aparentemente, al menos, se procederá a la traducción. En el Spectrum de 16k el ensamblaje comienza al ejecutar el comando directo: RANDOMIZE USR

275ØØ (para la versión de 48k la dirección es 6ØØØØ), y, si todo va bien, aparecerá en la pantalla un listado del código máquina obtenido, con las direcciones en decimal y los códigos de operación en hexadecimal, junto a las instrucciones originales. El ensamblador se ejecuta dos veces, haciendo que la pantalla parpadee si el programa fuente era corto. El motivo de esto es que la primera vez que examina el código fuente, el ensamblador puede encontrar nombres de etiquetas que se definen más tarde en el programa, y a las que, por tanto, no es posible asignar una dirección. Simplemente, se toma nota de ellas y en el segundo paso a través de todo el programa se resuelven todas esas "referencias adelantadas", asignando las direcciones correctas a las etiquetas. Los programas pequeños ocuparán menos de una pantalla, y, si se desea una copia escrita en la impresora, se pulsa la tecla COPY del computador. Cuando el código ensamblado requiere más de una pantalla, la traducción se detendrá al llenarse aquella y continuará cuando se pulse alguna tecla. Durante el segundo paso, al llenarse la pantalla, se imprimirá ésta pulsando la tecla "p", artificio impuesto por el hecho de que la tecla COPY del Spectrum no se puede utilizar mientras esté ejecutándose el ensamblador.

Mensajes de error

Los errores se detectan y señalan perfectamente. Si existen incorrecciones en alguna de las órdenes "go", "org" o "finish", se imprime un mensaje de error, en vez de comenzar la traducción. Si el fallo se encuentra en la sintaxis de las instrucciones de lenguaje ensamblador, como sería escribir "lda,12" en lugar de "ld a, 12", el ensamblaje se detendrá en la línea errónea, y el mensaje de error especificará en detalle qué sentencia de la línea REM de BASIC es la que provocó la detención. También se escribirá uno de los mensajes de error usuales del sistema operativo del Spectrum, que será: "B integer out of range", o "2 variable not found", o "Q parameter error", pero eso carece de importancia: ¡es un efecto secundario!

Características útiles

Además de la instrucción "org" (estrictamente hablando, ésta es una *pseudoinstrucción*, es decir, no genera código máquina), el ensamblador ULTRAVIOLET permite otras cuatro de este tipo. Una de ellas es "equ", que asigna una dirección a una etiqueta que aún no ha aparecido. Así:

```
3Ø REM equ 2356Ø Bucle
```

hace que sea posible escribir "bucle" en cualquier lugar donde deba escribirse la dirección 23560. Esto resulta muy útil para crear diferentes versiones de un programa (tal vez una para 16k y otra para 48k, entre otras), ya que todas las direcciones se cambian corrigiendo simplemente unas cuantas sentencias equ. Los creadores del ULTRAVIOLET sugieren que éste puede utilizarse para enlazar secciones separadas de código, de forma que se ejecuten como un programa único, como sucede con el propio ensamblador en versión de 16k.

Las restantes pseudoinstrucciones que admite, son "defb", "defw" y "defs", que constituyen una especie de comandos POKE. La primera, "defb", coloca el octeto que le sigue en la posición de memoria donde se ensambla la instrucción; "defw" permite almacenar dos octetos en la memoria, primero el menos significativo. La pseudoinstrucción "defs" es una de las más valiosas del grupo, porque almacena una cadena de códigos ASCII en la memoria. Por ejemplo, "defs Sinclair", colocará los ocho códigos ASCII de las letras que componen ese nombre, en la memoria.

Otras características y resumen final

Es posible programar código en forma tal que se automodifique, o sea, que él mismo cambie sus propias direcciones y/o datos. Si se coloca una etiqueta señalando una instrucción de carga (de un octeto, o de una dirección), más adelante es factible escribir otra instrucción que altere el contenido de las posiciones "etiqueta + 1" y/o "etiqueta + 2", con lo que estaremos alterando el programa mismo. Esta técnica de programación es más avanzada de lo que corresponde a este libro de introducción, pero es una característica poderosa del ensamblador.

Una vez que se denominan las particularidades del ensamblador ULTRAVIOLET (que parecerán negativas a quien haya probado algún otro ensamblador), resulta una herramienta muy valiosa. No obstante, no es un ensamblador de la categoría del increíble ZEN, que se usa en los computadores TRS-80, Nascom y Sharp. Con él, los programadores de código máquina del ZX-81 se sentirán como en su casa, y es especialmente interesante en el Spectrum de 48k, que tiene mucha más memoria libre para experimentar con el código máquina. Los programas de este libro se escribieron antes de que se comercializara ULTRAVIOLET, aunque el programa de demostración que aparece en las Figuras 8.1 y 8.2, da una idea de las posibilidades de este ensamblador y de los listados que produce. Otro ensamblador, de Abersoft, funciona con los Microdrives, pero aún no está disponible.

```
10 REM

20 REM go
30 REM org 23760
40 REM xor a
50 REM ld de, Dire
60 REM call 3082
70 REM set 5, (iy + 2)
80 REM call 5588
90 REM ret
100 REM Dire; defb 128
105 REM defs, Pulse cualquier tecla...
110 REM defb 174
120 REM finish
```

Fig. 8.1. Forma de escribir las instrucciones en el ensamblador ULTRAVIOLET. Este programa se puede grabar como cualquier otro en BASIC. Para la traducción, debe ejecutarse el comando RANDOMIZE USR 27500 (en la versión de 16k).

```
org 23760
23760  AF                  xor a
23761  11  DF  5C          ld de, Dire
23764  CD  0A  0C          call 3082
23767  FD  CB  02  EE      set 5, (iy + 2)
23771  DC  D4  15          call 5588
23774  C9                  ret
Dire
defb 128
defs Pulse cualquier tecla...
defb 174
```

Fig. 8.2. Aspecto del programa traducido cuando se imprime en pantalla. El ensamblador escribe un mensaje y espera que usted pulse una tecla.

Capítulo 9

El final del recorrido

Uno de los problemas de escribir un libro sobre programación en código máquina es saber dónde detenerse. Se podrían escribir grandes volúmenes de programación para el Spectrum y aún quedaría materia para muchos más, por lo que cualquier punto final será bastante arbitrario. Mi intención ha sido presentar la programación en código máquina del Spectrum, de tal forma que el lector sea capaz de seguir su aprendizaje en libros mucho más especializados. Este capítulo cierra el libro, mencionando algunas instruciones adicionales e ilustrando el empleo de unas cuantas características más del Spectrum.

Empezaremos con otro conjunto de instrucciones del Z-8Ø: el grupo de desplazamiento de bloques. La frase "desplazamiento de bloques" resume en sí misma el propósito de este grupo, cuya utilidad es el movimiento del código desde una parte a otra de la memoria, colocando algunas direcciones de memoria en ciertos registros, y usando una única instrucción de lenguaje ensamblador. En la Figura 9.1 puede ver una lista de estos códigos con una breve explicación de lo que hace cada uno. Puesto que son tan parecidos unos a otros en su modo de operar, elegiremos uno de ellos, LDIR, por ejemplo, como muestra.

Mnemónico	Acción
LDI	Coloca el contenido de la dirección almacenada en HL en la dirección señalada por DE, incrementa HL y DE, decrementa BC.
LDIR	Como LDI, pero repite la acción hasta que BC contiene cero.
LDD	Como LDI, pero los pares HL y DE de registros se decrementan.
LDDR	Como LDD, pero repite la acción hasta que BC contiene cero.

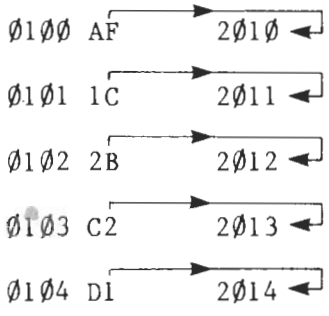
Fig. 9.1. Los comandos de desplazamiento de bloques del Z-80.

LDIR es la abreviatura (inglesa) de carga, decremento e incremento de registro. El comando utiliza los registros dobles HL, DE y BC. El par HL contiene la dirección de comienzo del bloque de memoria "fuente", DE la dirección de comienzo del bloque de "destino", y BC almacena el número de octetos de código que se van a transferir desde el bloque fuente al bloque de destino. Cuando se ejecuta la instrucción LDIR, se copia un octeto de la dirección señalada por HL en la dirección señalada por DE, el registro contador BC se *decrementa,* y las direcciones contenidas en los pares HL y DE se *incrementan.* Este proceso continúa hasta que BC llega a cero. La Figura 9.2 describe la acción comentada.

Imagine que la dirección contenida en HL es 235ØØ, que la dirección almacenada en DE es 236ØØ, y que BC contiene 5.

La operación comienza copiándose el octeto de la dirección 235ØØ en la dirección 236ØØ, después se incrementan ambas, pasando a ser 235Ø1 y 236Ø1, el par BC se decrementa y su contenida es ahora un 4. Se repite la transferencia, copiándose el octeto de 235Ø1 en 236Ø1, y BC contiene 3 después del decremento, El proceso terminará cuando el número almacenado en BC sea cero y se haya copiado 5 octetos.

Otro ejemplo



HL ØlØØ
DE 2ØlØ
BC ØØØ5

LDIR efectúa la transferencia de octetos desde las direcciones que comienzan en ØlØØ, a las direcciones cuyo comienzo es 2ØlØ, como indican las flechas.

NOTA: Todos los números en hexadecimal.

Fig. 9.2. Funcionamiento de LDIR.

La Figura 9.3 muestra una aplicación de la instrucción LDIR, que produce una imagen en la pantalla del Spectrum. El número que se carga en el par HL de registros es el comienzo del fichero de pantalla del Spectrum, lo que implica que, al almacenar cualquier cosa en esas direcciones, aparecerá algo en la pantalla. El par DE se carga con la dirección siguiente a la del par HL, y en BC se almacena el número de octetos que se desplazarán, que, en este caso, resulta ser la diferencia entre las direcciones de final y principio del fichero de pantalla, menos uno. Después se almacena en la primera dirección, la que se encuentra en HL, el octeto 68, correspondiente al número binario Ø1ØØØ1ØØ.

```
VIDEO EQU 16384
LONG  EQU 6143
COM:  LD HL, VIDEO
      LD DE, VIDEO + 1
      LD BC, LONG
      LD (HL), 68
      LDIR
      RET

1Ø CLEAR 325ØØ: LET j = 325ØØ
2Ø FOR n = Ø TO 13 : READ b
3Ø POKE j + n, b : NEXT n
4Ø LET z = USR j
1ØØ DATA 33, Ø, 64, 17, 1, 64, 1,
255, 23, 54, 68, 237, 176, 2Ø1
```

Fig. 9.3. Uso de LDIR para producir una imagen en la pantalla. ¡Pruébelo para ver su velocidad!

Cuando se empieza a ejecutar LDIR, el octeto 68 se copia de la dirección 16384 en 16385. Luego se decrementa el par BC, y se incrementan HL y DE, cuyos nuevos contenidos serán 16385 y 16386, respectivamente. En la siguiente iteración, el octeto que se había colocado en 16385 se copia en 16386, y continúa esta cadena de copias en posiciones sucesivas, hasta que todo el fichero de pantalla se rellene con el octeto 68. Traduzca esto a octetos, por sí mismo, y observe cómo se ejecuta; es una poderosa demostración de lo rápido que es el código máquina para cosas de este tipo. Esa es la razón de que los juegos de acción rápida se programen siempre en este lenguaje. Se puede borrar la pantalla y volver a ejecutar el programa, tecleando LET x = USRj, o RANDOMIZE USRj. Si emplea PRINT USR J, aparecerá el número Ø en el extremo superior izquierdo de la pantalla.

estropeando de algún modo el efecto. En caso de que prefiera una rejilla más estrecha, pruebe a poner el número 85 en lugar de 68; la forma de la imagen depende de la secuencia de unos y ceros existentes en la versión binaria del número, ya que cada 1 corresponde a un punto negro, y cada Ø a uno blanco. Es divertido escribir un programa BASIC que haga lo mismo que éste, o sea, poner un octeto en cada posición de memoria del fichero de pantalla, y comparar el tiempo que tardan ambos en ejecutarse.

Paso de valores al código máquina

Ya vimos como el sistema operativo del Spectrum está diseñado de forma que el valor almacenado en el par BC de registros, al final de una rutina en código máquina, puede ser devuelto a BASIC cuando se ejecuta un comando RET. Sin embargo, el manual del Spectrum no dice nada sobre el paso de valores desde BASIC al programa en código máquina. Otros ordenadores que cuentan también con la instrucción USR, permiten pasar a las rutinas en código máquina un valor que es el argumento colocado a continuación del nombre USR, pero el Spectrum interpreta ese número como la dirección donde se encuentra el código máquina que se va a ejecutar. La ventaja de esta última clase de función USR es que resulta muy fácil averiguar lo que hacen las diversas rutinas de la memoria ROM: Unicamente hay que teclear RANDOMIZE USR dirección, siendo esa dirección la de comienzo de la subrutina que se quiere ver funcionar. Luego, se pulsa ENTER y ¡a ver que ocurre!

Un método bastante hábil consiste en pasar un valor almacenado en una variable BASIC. Si tenemos una variable n, por poner un ejemplo, en la tabla de variables existirá un elemento correspondiente a aquella, que será análogo al que aparece en la Figura 9.4. Suponiendo que nos limitamos a los números enteros positivos, entonces el valor de la tabla de variables puede pasarse fácilmente a una rutina en código máquina, siempre que ésta pueda localizar ese valor. Se puede conseguir esto último, gracias a que el comienzo de la tabla de variables se encuentra almacenado en la dirección 23627 de la memoria RAM. Veamos un programa muy simple (aunque muy limitado).

Elemento de la tabla de variables correspondiente a LET n = 5

11Ø	Ø	Ø	5	Ø	Ø
Principio					Fin

Fig. 9.4. Elemento de la TDV para la variable n (un repaso).

La rutina de búsqueda tiene que tomar los octetos contenidos en 23627 (menos significativo) y en 23628 (más significativo), y reunirlos para formar una dirección, que es precisamente la del principio de la tabla de variables. Llevando esa dirección al par de registros HL, podremos usar uno de los comandos de "búsqueda e informe" del Z-8Ø para encontrar el nombre de la variable "n" que, por ser una variable numérica simple, se codifica en ASCII normalmente, es decir, como 11Ø. El comando de búsqueda e informe es CPIR, que pertenece al grupo de instrucciones de búsqueda de bloques resumido en la Figura 9.5. CPIR buscará a través de un bloque específico de la memoria,

Mnemónico	Acción
CPI	Comparar el octeto almacenado en la dirección señalada por HL con el contenido de A. Si ambos coindicen, el indicador Z se pone a "1" y, en otro caso, a "Ø". Después se incrementa el par HL y se decrementa BC.
CPIR	Como CPI, pero si los octetos no coinciden se repite la acción hasta que BC llegue a cero.
CPD	Como CPI, pero después de la comparación, el registro doble HL se decrementa.
CPDR	Es la versión automática de CPD.

Fig. 9.5. Comandos de búsqueda de bloques del Z-80.

examinando octeto por octeto, hasta encontrar el valor deseado. La dirección del principio del bloque se guarda en HL, el valor que se busca se almacena en A, y el número máximo de octetos que se deben examinar se coloca en BC. Cuando se encuentra el código de operación de CPIR (que es de dos octetos), se compararán los contenidos del acumulador y de la dirección que está en HL, y, si no coinciden, se incrementan HL y se decrementa BC, continuando la búsqueda hasta que se encuentre un octeto de valor igual al contenido de A, o el par BC llegue a cero.

En la figura 9.6 aparece el organigrama para este programa. No usaremos para nada el valor de la variable en este ejemplo, ya que, de momento, nos interesa más ver cómo se busca dicho valor, que hacer algo con él una vez encontrado. Las partes clave del diagrama de flujo son: obtener la dirección de la tabla de variables, la búsqueda de la dirección donde se encuentra el nombre de la variable y, por último, llegar a la dirección en la que está almacenado el valor de aquella.

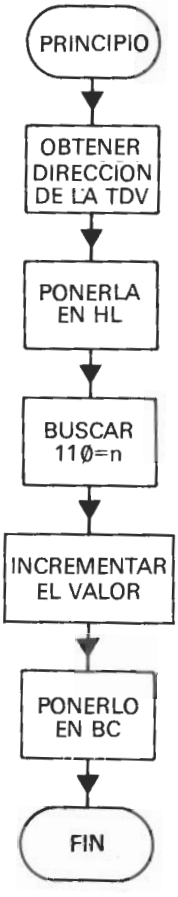


Fig. 9.6. Organigrama para encontrar un nombre de variable.

Suponiendo de nuevo que trabajamos con números enteros positivos menores que 65535, el valor de la variable consta de dos octetos solamente, que se cargan en el par de registros BC, para poder imprimir el número en la pantalla y asegurarnos de que lo hemos encontrado.

La instrucción CPIR dejará el valor del par HL incrementado, con lo cual la dirección contenida en dicho par apunta el octeto que sigue al nombre de la variable, cuando se haya encontrado ésta. Así pues, sólo necesitamos dos incrementos más, para que la dirección almacenada en HL coincida con la del octeto de menor peso de los dos que componen el valor de la variable en cuestión.

El programa de la Figura 9.7 es la versión en lenguaje ensamblador del organigrama anterior. Se carga el par HL de registros con la dirección de RAM en la que está contenido el octeto manos significativo de la posición de comienzo de la tabla de variables. Ese octeto se

```
LD HL, 23627   ;apuntador a la tabla de variables
LD E, (HL)     ;obtener el octeto de menor peso
INC HL         ;incrementar el apuntador
LD D, (HL)     ;obtener el octeto de mayor peso
LD BC, FFFFH;contador cargado con el máximo número posible
EX DE, HL      ;llevar la dirección de la tabla de variables a HL
LD A, 11Ø      ;octeto buscado
CPIR           ;búsqueda del octeto
INC HL         ;saltar octeto...
INC HL         ;dos veces para que HL apunte al valor
LD C,(HL)      ;llevar a C el octeto del menor peso
INC HL         ;apuntar a octeto de mayor peso
LD B, (HL)     ;octeto superior llevado a B
RET            ;volver a BASIC
```

Fig. 9.7. Programa en lenguaje ensamblador para encontrar el nombre de una variable.

pasa al registro E, y se incrementa HL, que contendrá ahora el valor 23628. El octeto almacenado ahí, el más significativo de la dirección, se lleva al registro D. Con eso, tendremos en DE la dirección de la tabla de variables y además en orden correcto: el octeto de mayor peso en D, y el de menor peso en E. Después de una o dos instrucciones, aparece el comando "EX DE,HL", que no se ha comentado hasta ahora. Como tal vez haya adivinado ya, su función consiste en intercambiar el contenido de los pares de registros DE y HL; tras ejecutarla, en HL se encontrará la dirección de la tabla de variables, y en DE, 23628, que ya no nos sirve para nada. Este comando de intercambio se emplea exhaustivamente para cargar en HL una dirección que hemos formado en otro par de registros, ya que siempre intentaremos que las direcciones importantes estén en HL, en lugar de en otro par cualquiera de registros. En este ejemplo, hemos llevado a BC el máximo número posible de octetos que se pueden contar. En una situación práctica, podría ser más sensato limitar la cuenta a cantidades menores, para evitar búsquedas largas de nombres de variables inexistentes. Una mejora adicional sería imprimir un mensaje de error, si no se encuentra la variable. Los mensajes de error se activan poniendo el número del código del error en la posición de memoria siguiente a una orden RST 8. Por tratarse de un ejemplo, no obstante, aquí se ha utilizado la máxima longitud posible para BC (¡esto evita también que el programa vaya más allá de los límites de un ejercicio de iniciación!) Recuerde que aunque alterase el valor inicial del par BC de registros, tendría que cargar en éste dos octetos. Por lo tanto, si desea que sólo se examinen

1ØØ octetos, es preciso escribir 1,1ØØ,Ø, donde 1 es el código de LD BC,NN, y los dos octetos siguientes corresponden al valor almacenado en BC al comenzar la búsqueda.

Posteriormente, se emplea CPIR para buscar un octeto de valor igual a 11Ø. Eso hace que el programa sea vulnerable, debido a que si la tabla de variables posee muchos elementos delante del que se destina a n, y uno cualquiera de ellos contiene el número 11Ø (tal como LET s = 11Ø, o el menos evidente LET s = 28165, que es 5 + 256*11Ø), entonces lo que se encuentra es la dirección de ese valor, no el de variable n. Es posible soslayar esta dificultad, contruyendo un programa mucho más elaborado; pero nuestra intención aquí es ilustrar los principios de la programación en código máquina, no hacer ostentaciones. De todos modos, si quiere una pista, una buena es fijarse en los tres bits más significativos del nombre de la primera variable, y aprovechar esa información para pasar directamente a la siguiente, y así sucesivamente. Esto no puede conseguirse con CPIR, motivo por el que he preferido exponer el método simple del ejemplo.

Una vez que se encuentra la dirección, recordando que HL apunta ahora a la posición inmediatamente superior a la del nombre de la variable, podemos incrementar dos veces HL, para obtener el octeto de menor peso del valor de n. Este se pone en C, se incrementa de nuevo HL, y el octeto siguiente, que será el de mayor peso, se pasa a B. Al retornar a BASIC, tendremos en BC el valor que buscábamos.

El programa listado en la Figura 9.8. carga en memoria la rutina, la ejecuta y presenta el resultado en la pantalla. En la línea 2Ø, la asignación LET n = 24Ø coloca este valor en la tabla de variables y la rutina USR, llamada en la línea 4Ø, encontrará dicho valor. La varia-

```
  1Ø CLEAR 325ØØ
  2Ø LET n = 24Ø
  3Ø GO SUB 1ØØØ
  4Ø PRINT USR j
  5Ø GOTO 9999
1ØØØ LET j = 325ØØ
1Ø1Ø FOR x = Ø TO 19: READ b
1Ø2Ø POKE j + x, b: NEXT x
1Ø3Ø RETURN
1Ø4Ø DATA 33, 75, 92, 94, 35, 86, 1, 255,
     255, 235, 62, 11Ø, 237, 177, 35, 35,
     78, 35, 7Ø, 2Ø1
```

Fig. 9.8. Versión del programa BASIC que almacena y ejecuta la rutina de búsqueda de variables.

ble n puede tomar cualquier valor entre Ø y 65535, pero se deben evitar los números negativos.

La utilidad de este procedimiento no se reduce a pasar el valor de una variable numérica simple. Puesto que el programa encuentra la dirección del nombre de una variable en la TDV, con unas cuantas modificaciones se puede usar para encontrar los nombres de variables de cadena y de tablas numéricas o de cadena, suponiendo que usted recuerde cómo se codifican los mismos (vea el Capítulo 2). Por ejemplo, si se trata de una tabla numérica, el nombre de la misma va seguido de un contador. Este último se carga en BC, y se establece un bucle para obtener los valores y transferirlos a direcciones de memoria RAM contenidas en el par de registros DE. En el análisis del nombre de una tabla que aparece en la Figura 9.9, vemos que la longitud de dicha tabla se almacena en dos octetos, a continuación se coloca el número de dimensiones y, tras él, el tamaño de la primera dimensión en dos octetos más. Si conseguimos tener en HL la dirección del octeto siguiente al tamaño de la primera dimensión, este registro estará seña-

La tabla se denomina "a", los valores van de 1 a 10, de manera que a(1) = 1, a(2) = 2, ..., a (10) = 10.

El elemento correspondiente de la tabla de variables es:

129	53 Ø	1	1Ø Ø	Ø Ø 1 Ø Ø
"a"	longitud	dim	1.ª dim	primer valor

Ø Ø 2 Ø Ø	.....	Ø Ø 1Ø Ø Ø
2.º valor		último valor

DIM significa dimensión de la tabla: cuántos grupos de números se ponen entre paréntesis; por ejemplo, a(4) es una tabla de una sola dimensión, b(2,3) es una tabla de dos dimensiones, etc...
La longitud comprende el número total de octetos que se encuentran detrás de los dos que componen la propia longitud, esto es, desde el octeto de la 1.ª dimensión hasta el último elemento de la tabla.

Fig. 9.9. Elemento típico de la tabla de variables correspondientes a una tabla numérica de una sola dimensión.

lando el principio de la tabla, que se encuentra tres octetos más allá del contador antes citado (reste 3 de BC, para obtener el número de octetos que deben transferirse realmente). Con esto, se pueden leer después los valores de los elementos, por medio de la rutina mostrada en el organigrama de la Figura 9.6.

Incrementando DE cada vez que se lee un número, los valores se almacenan en direcciones consecutivas de memoria, lo cual permitirá su posterior utilización en rutinas de clasificación, entre otras, que quedan fuera del alcance de este libro.

También se puede acceder del mismo modo a los elementos de la tabla de variables asignados a tablas de cadenas.

Caballos de carreras

Una gran cantidad de libros de programación en código máquina dedican bastante espacio a las rutinas numéricas (sumas, restas, productos y divisiones). En general, es un esfuerzo innecesario, porque esas rutinas existen ya en la memoria ROM del Spectrum, y pueden llamarse fácilmente a través de BASIC. El hecho de que se ejecuten con mayor lentitud en este último lenguaje, es muy pocas veces una desventaja real. Por tanto, cualquier programa que requiera muchos cálculos aritméticos se escribirá mejor en BASIC, a no ser que resulte tan lento que esa opción sea inviable (algunos programas de "hojas de cálculo electrónicas" pertenecen a esta categoría). El código máquina se hace verdaderamente indispensable cuando usted quiere una gran velocidad, tal vez para manejar gráficos, o para llevar a cabo acciones que normalmente no proporciona BASIC, como enviar octetos a una impresora en serie, a través del puerto del magnetofón. Tendrá que decidir, en alguna etapa del desarrollo de un programa, si debería escribir éste completamente en BASIC, o principalmente en BASIC con algunas rutinas en código máquina, o totalmente en código máquina. Si no dispone de un ensamblador, yo le aconsejaría que no tratase de escribir el programa completo en código máquina, salvo que sea muy corto.

Así pues, muchos programas se realizan de forma óptima, alternando secciones en BASIC con rutinas en código máquina. El método de acceder al código máquina por medio de la función "USR dirección", que proporciona el Spectrum, corrobora lo que hemos expuesto; en resumen: es una buena decisión aprovechar esta característica del computador.

En muchos programas que manejan gráficos, sólo aquellas partes que exigen movimientos rápidos, o requieren rellenar la pantalla (o una

sección de ésta) con gran velocidad, necesitan escribirse en código máquina. Cabe la posibilidad de tener en memoria RAM varios bloques de código, reservando previamente el espacio adecuado; después se puede llamar uno u otro, poniendo la dirección correspondiente a continuación del comando USR. De esta manera, su programa BASIC dispondrá de tantas rutinas en código máquina como usted desee.

El paso de variables a las subrutinas en código máquina (veremos más tarde otro procedimiento) permite intentar acciones como rellenar la pantalla a partir de un lugar especificado por el valor de otra variable (por éste último sirve como contador en un bucle de retardo del tipo DJNZ).

El código máquina puede llegar a ser fascinante, y es muy tentador lanzarse a crear programas muy largos, solamente para convencerse uno mismo de que se domina ese lenguaje. Esa tentación debe evitarse, a menos que se tenga mucho tiempo y un gran empeño en hacerlo. Si su objetivo es, simplemente, producir programas eficientes, la mejor apuesta es, probablemente, emplear una mezcla de BASIC y código máquina. Si, en cambio, su vocación es crear programas de juegos de acción rápida, quizá sea más provechoso tener en cuenta algún lenguaje alternativo a BASIC, como FORTH, que trabajar directamente con código máquina. ¡Elija el caballo correcto para la carrera, y será usted un ganador!

El último programa

Estamos casi llegando al final de este libro, lo que significa que puede colgar su diploma de "Licenciado en código máquina", y empezar a consultar algunos de los libros que se citan en el Apéndice A. Sin embargo, antes de terminar le dejaremos un último programa muy útil. Sirve para encontrar la dirección de almacenamiento en memoria de cualquier línea BASIC. Hemos dicho que es muy útil, porque en muchas otras rutinas de ayuda a la programación (como las que renumeran las líneas de los programas BASIC), se utilizan las mismas técnicas que veremos en este ejemplo. Además, es el programa más elaborado de los que aparecen en este libro, ¡y el único que me hizo desear que los programas ensambladores hubiesen estado antes disponibles!

He utilizado, como rutina auxiliar de entrada, el programa que desarrollé para cargar en BC el valor de una variable entera cuyo nombre es "n". Evidentemente, éste no es el único medio de poner este valor en un sitio adecuado. Después veremos un procedimiento mucho más simple; pero quiero mostrar cómo pueden unirse dos subrutinas para formar un programa más extenso. Recordemos que la rutina de

búsqueda de variables pone el valor de la variable entera en el registro doble BC. Como los números de línea son siempre enteros positivos, usaré la rutina tal y como está, ya que servirá a la perfección para nuestros propósitos. La etapa siguiente, entonces, es diseñar un programa que recorrerá las líneas una por una, comprobando sus números hasta que encuentre uno que coincida con el valor contenido en BC. No nos preocupa, de momento, lo que haga el programa si no encuentra el número de línea buscado.

Seguiremos el organigrama de la Figura 9.10. Lo primero es encon-

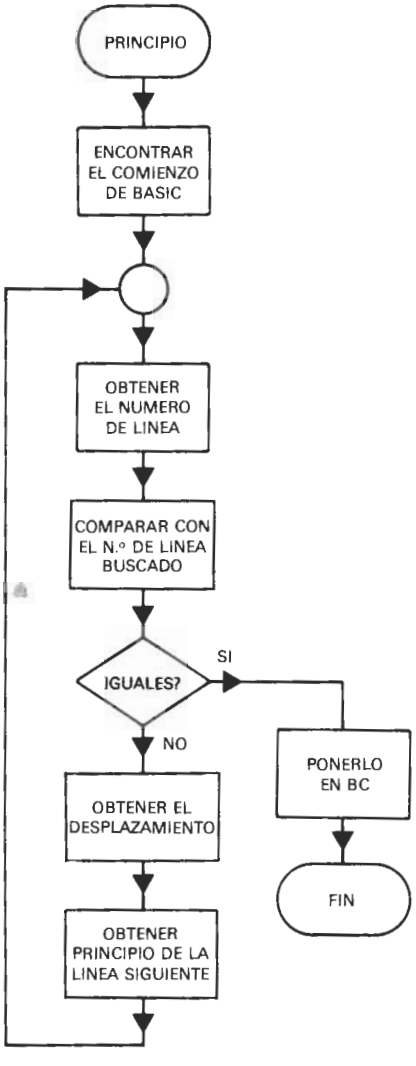


Fig. 9.10. Búsqueda de la dirección de comienzo de una línea de un programa BASIC: el organigrama.

trar la dirección de comienzo del programa BASIC. Esto se hace colocando en el par HL dicha dirección, que está almacenada en la zona reservada de la memoria RAM. Una vez hecho esto, sabemos que los dos primeros octetos de código corresponden al número de la primera línea, aunque están colocados en orden contrario al habitual: el más significativo seguido del menos significativo. Podemos extraerlos y compararlos con los octetos del registro BC, que son los que corresponden al número que buscamos (y que fueron colocados por la rutina de búsqueda de variables). La comparación tendremos que hacerla en dos pasos, pero el organigrama no debe entrar en esos detalles. Si los octetos coinciden, se pasa el contenido de HL a BC, y el programa devuelve el control a BASIC, para que se imprima el resultado en la pantalla. En caso contrario, hay que leer los dos octetos siguientes de la línea. Estos constituyen un "desplazamiento", que nos dará la dirección del final de la misma. La línea siguiente comienza en la posición inmediatamente superior y, por tanto, tras incrementar HL, podemos repetir todo el proceso de búsqueda.

Debido a que el organigrama en este ejemplo no especifica demasiado las acciones, la correspondencia entre éste y el programa en lenguaje ensamblador no será tan estrecha como lo era en ejemplos anteriores. Cuando no esté seguro de cómo se desarrolla cualquier elemento simple de un organigrama, es una buena idea dibujar un diagrama de flujo solamente para ese elemento. Esto es lo que hemos hecho en la figura 9.11, en la que aparecen las etapas "COMPARA" y "¿COINCIDEN?", del organigrama general. La justificación de esto es que se precisan dos comparaciones. Mediante la primera tratamos de averiguar si el octeto de mayor peso del número de línea coincide con el que se encuentra en el registro B; la segunda comparación nos permite comprobar si el octeto menos significativo del número de línea es igual al octeto almacenado en el registro C. Si falla cualquiera de ellas (o ambas), habrá que ir a la rutina "OBTENER DESPLAZAMIENTO". En caso de ser ciertas las dos, se copiarán en el par BC los octetos de la dirección, y el control volverá a BASIC.

En la Figura 9.12 tenemos la versión en lenguaje ensamblador totalmente comentada, y con los códigos decimales añadidos en la columna de la derecha. La dirección del "apuntador" es 23635; tomando los octetos de 23635 y 23636, obtenemos la dirección de comienzo de BASIC en el registro DE (esencialmente, es la misma clase de programación empleada en la rutina de búsqueda de variables). Intercambiando DE y HL, esa dirección pasa a este último par. No olvide que el registro doble HL es el más apropiado para nuestros fines, debido a que dispone de un rango mayor de operaciones, incluyendo las de búsqueda y comparación de bloques, que no se utilizan aquí.

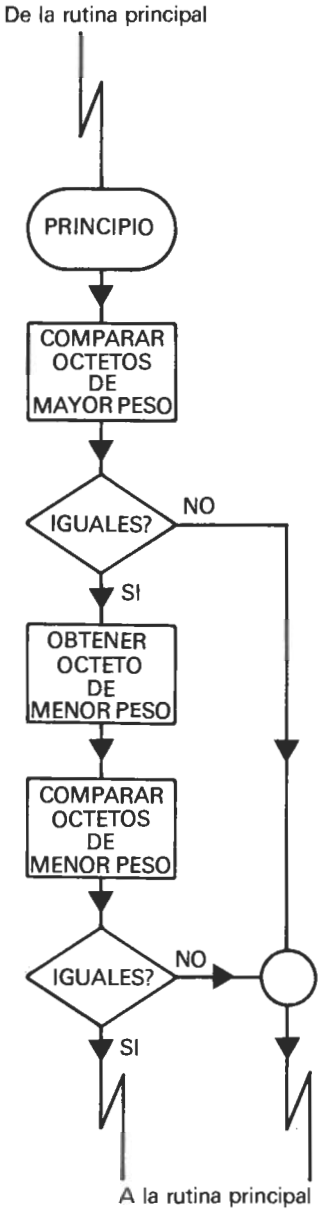


Fig. 9.11. Un organigrama más detallado de las etapas de comparación.

Los programadores suelen utilizar HL para las direcciones de origen más importantes, y DE para las direcciones de destino y para construir direcciones que pasarán luego a HL y el par BC como contador.

Los dos primeros octetos del área de programa BASIC, son los que componen el primer número de línea. Cargando en A el octeto de la dirección señalada por HL, conseguimos el octeto de más peso del

(Inicialmente, BC contiene el número de línea obtenido por la subrutina anterior)

```
        LD HL, 23635   ;obtener apuntador          33, 83, 92
        LD E, (HL)     ;obtener octeto inferior    94
        INC HL         ;incrementar apuntador      35
        LD K, (HL)     ;obtener octeto superior    86
        EX DE, HL      ;comienzo de BASIC          235
Comp:   LD A, (HL)     ;octeto de mayor peso
                        del n.º de línea           126
        CP B           ;¿iguales?                  184
        INC HL         ;incrementar apuntador      35
        JR NZ, Sig     ;distintos                  32, 8
        LD A, (HL)     ;octeto de menor peso       126
        CP C           ;comparación                185
        JR NZ, Sig     ;distintos                  32, 4
        DEC HL         ;línea encontrada           43
        LD C, L        ;cargar en C                77
        LD B, H        ;y B                        68
        RET            ;retorno a BASIC            2Ø1
Sig:    INC HL         ;octeto de menor
                        peso del
                        desplazamiento: llevarlo   35
        LD E, (HL)     ;a registro E               94
        INC HL         ;incrementar apuntador      35
        LD D, (HL)     ;octeto mayor peso a D      86
        ADD HL, DE     ;fin de la línea            25
        INC HL         ;principio de línea         35
        JR Comp        ;repetir la búsqueda        24, 235
```

Fig. 9.12. Versión en ensamblador del buscador de líneas. ¡Recuerde que debe ir precedido por la rutina de búsqueda de variables!

número de línea y lo comparamos con el correspondiente de la línea buscada, es decir, con el octeto de registro B, no sin antes haber incrementado HL para evitar futuras complicaciones. La operación "CP B" afecta a los indicadores del registro de estado y, por lo tanto, colocamos inmediatamente detrás el salto "JR NZ", que cederá el control a la rutina "encontrar la línea siguiente", si los octetos son distintos.

Muy a menudo, coincidirán los octetos de mayor peso; por ejemplo, cuando el programa BASIC no tenga números de línea mayores que 255 y el número buscado sea, a su vez, menor o igual que 255. En

tales casos es necesario comprobar los octetos de menor peso y, para ello, se sigue el mismo método de poner la instrucción "CP C", y a continuación "JR NZ" para saltar a la rutina "encontrar la línea siguiente", si los números no coinciden. Cuando ocurre lo contrario (ambos octetos son iguales), el par HL debe decrementarse, a fin de que apunte de nuevo al principio de la línea, con lo que se pueden transferir los octetos de H y L a B y C. Por último, se devuelve el control al programa BASIC, pasándole la dirección encontrada a través del par BC. Si no se ha encontrado la línea, la rutina que empieza en la etiqueta "Sig" obtendrá el principio de la línea siguiente. Se incrementa el valor de HL, de modo que apunte a la dirección del octeto bajo del desplazamiento, y éste se copia en el registro E; luego se incrementa otra vez HL y el octeto correspondiente a la nueva dirección se lleva a D. Sumando HL a DE, e incrementando el resultado (ya que DE + HL da la dirección del final de la línea actual), conseguimos la dirección de la línea siguiente y podemos pasar a repetir la búsqueda, saltando a la etiqueta "Comp".

Cuando se escriben programas de la longitud del anterior (más o menos), ¡se empieza a echar de menos un ensamblador! Es bastante aburrido ensamblarlo "a mano", y mucho más aún modificarlo. Puede decidir si le gusta o no, traduciéndolo y probando a modificarlo de algún modo.

Antes de finalizar, me gustaría señalar una drástica simplificación referente a la colocación del número de línea que se va a buscar en BC. Tal y como estaba el programa hasta ahora, usaba la rutina de búsqueda de variables, pero este método presenta inconvenientes. Uno de sus principales defectos es que nos obliga a que "n" sea la primera variable en la tabla de éstas, en caso de que en cualquier parte de dicha tabla aparezca el mismo número. Teóricamente, podríamos incluir "GOSUB 1000" como primera línea del programa BASIC, colocar el código en la memoria una sola vez, y después utilizarlo. Esto aceleraría bastante la ejecución, porque, tal como aparece en la Figura 9.13, vuelve a poner el código en memoria cada vez que se ejecuta, lo que desperdicia tiempo. En cambio, si llamamos a la subrutina cargadora de código una única vez, la tabla de variables tendrá elementos por delante del que corresponde a n, y eso es arriesgado.

El programa de la Figura 9.14 emplea otro método. Eliminando la rutina de búsqueda de variables, reducimos bastante la longitud del código máquina, de manera que hay que almacenar muchos menos octetos con POKE, y modificando el programa BASIC podemos acelerar la acción. El programa se puede almacenar en memoria, leyéndolo del magnetofón si se desea, o por medio de una subrutina como antes; en ambos casos, debe hacerse antes de ejecutarse la sentencia INPUT

```
  5 CLEAR 325ØØ
 1Ø INPUT "Numero de linea? -"; n: LET n
    = INT n
 2Ø GO SUB 1ØØØ
 3Ø LET x = USR j: PRINT "La linea esta en -"; x
 4Ø GOTO 9999
1ØØØ LET j = 325ØØ
1Ø1Ø FOR x = Ø TO 46: READ b
1Ø2Ø POKE j + x, b: NEXT x
1Ø3Ø RETURN
1Ø4Ø DATA 33, 75, 92, 94, 35, 86, 1, 255,
     255, 235, 62, 11Ø, 237, 177, 35, 35, 78,
     35, 7Ø
1Ø5Ø DATA 33, 83, 92, 94, 35, 86, 235, 126,
     184, 35, 32, 8, 126, 185, 32, 4
1Ø6Ø DATA 43, 77, 68, 2Ø1, 35, 94, 35, 86,
     25, 35, 24, 235
```

Fig. 9.13. El programa puesto en forma BASIC. Tiene la desventaja de ser lento, porque se carga el código máquina en cada ejecución.

de la parte BASIC. El código máquina comprende ahora la instrucción "LD BC, dirección" (3 octetos), seguida del resto de la rutina buscadora de líneas, inalterada. Dado que la rutina empieza en 325ØØ, en este ejemplo, los números que se cargan en BC al principio de esta sección ocupan las posiciones 325Ø1 y 325Ø2. Si el número de línea es menor que 256, es posible almacenarlo directamente en 325Ø1, porque hemos puesto los valores 1Ø y Ø en el código máquina inicial. Si el número es mayor que 255, hay que dividirlo en dos octetos (de mayor y de menor peso), y llevar éstos a las direcciones apropiadas con POKE. Con esta técnica de colocar un valor variable, puede examinar las líneas de un programa BASIC. Comience ejecutando el comando CLEAR 325ØØ en modo directo. Después cargue el código máquina con el magnetofón; cargue el programa BASIC que desea analizar, y luego añada la parte del programa de la Figura 9.14 que ejecuta la rutina en código máquina (líneas de 4Ø a 9Ø), con números de línea superiores a los del programa que se va a examinar. Con un comando directo GOTO (comienzo del programa BASIC buscador de líneas), se ejecuta la parte BASIC del buscador de líneas, que llamará al código máquina cuando sea preciso, colocando en la memoria con el comando POKE el número de línea seleccionado, activando el código máquina e imprimiendo el resultado en pantalla.

El código máquina del programa de búsqueda de líneas, pero sin la rutina de búsqueda de variables numéricas. La primera instrucción es LD BC, 1Ø (código 1, 1Ø, Ø), que carga BC con el número de línea 1Ø por defecto, es decir, si ningún otro número de línea se le pasa como parámetro. El programa BASIC almacena el código máquina y, mediante por defecto, es decir, si ningún otro número de línea se le pasa como parámetro. El programa BASIC almacena el código máquina y, mediante por defecto, es decir, si ningún otro número de línea se le pasa como parámetro. El programa BASIC almacena el código máquina y, mediante POKE, coloca en memoria el número de línea, de manera que éste se lleva a BC cuando se ejecuta el código máquina. Esto permite que la parte de búsqueda de la rutina trabaje en un bucle, o sea, se puede cambiar la línea 9Ø por GOTO 4Ø, si es necesario.

```
  1Ø CLEAR 325ØØ: LET j=325ØØ
  2Ø FOR n=Ø TO 3Ø: READ b
  3Ø POKE j+n, b: NEXT n
  4Ø INPUT "Numero de linea? -"; n: LET n
     =INT n
  5Ø IF n<256 THEN POKE 325Ø1, n
  6Ø IF n>=256 THEN LET m=INT (n/256): POKE
     325Ø2, m: POKE 325Ø1, n-256*m
  7Ø LET x=USR j
  8Ø PRINT "La linea esta en -"; x
  9Ø GOTO 9999
1Ø5Ø DATA 1, 1Ø, Ø, 33, 83, 92, 94, 35, 86,
     235, 126, 184, 35, 32, 8, 126, 185, 32,
     4
1Ø6Ø DATA 43, 77, 68, 2Ø1, 35, 94, 35,
     86, 25, 35, 24, 235
```

Fig. 9.14. Un buscador de líneas más rápido. El número de línea se pone directamente en el código máquina con POKE, en vez de usar el buscador de variables. Esto permite almacenar el código máquina una vez, y luego utilizarlo repetidamente.

Tal vez podría gustarle escribir ahora un programa de búsqueda de líneas en BASIC, que utilice PEEK y POKE, para ver cuánto tarda en realizar la misma tarea (otra demostración del valor del código máquina). Después de eso, puede empezar a buscar problemas que necesiten solución: ¡usted ya no es un principiante!

Apéndice A

Libros y revistas

La informática es una disciplina que cambia muy rápidamente, y continuamente aparecen nuevas ideas en este campo. Unicamente se puede estar al día, teniendo una información exhaustiva de los avances logrados. Es imprescindible unirse a un grupo de usuarios, porque los miembros de tales grupos exploran con rapidez todas las posibilidades que ofrece el computador que hayan elegido, e intercambian sus conocimientos unos con otros. El segundo punto es hacer uso de las revistas. Como antes dije, *Personal Computer World* publica una serie sobre rutinas en código máquina, y también encontrará cosas muy interesantes en revistas como *Computing Today, Electronics and Computing Monthly* y *Your Computer*.

Los libros siguientes le resultarán muy valiosos a medida que vaya adquiriendo una mayor experiencia: Z-8Ø Assembly Language Programming, de L. A. Leventhal; se trata de un extenso estudio sobre el ensamblador del Z-8Ø, que sirve a un tiempo como guía de referencia y como orientación del estilo de programación. Si se toma en serio la programación del Z-8Ø, este libro es indispensable.

Z-8Ø CPU Instruction Set, publicado por SGS-ATES; es una lista completa de todas las instrucciones del Z-8Ø: sus códigos y sus acciones. No es fácil de leer, y algunas veces resulta frustrante a causa de la estructura que tiene; sin embargo, es un libro sin el cual yo no quisiera estar. Quizá le guste saber que Alan Tootill, el autor de la serie que he mencionado de la revista PCW, está escribiendo con David Barrow un libro de código máquina del Z-8Ø. Se llamará; *Z-8Ø Machine Code for Humans*. La intención de los autores es demostrar que el código máquina, ¡puede ser divertido!

Apéndice B

Números en coma flotante

Se ha expuesto ya, en el texto, la codificación de números enteros; este apéndice, que es sólo para los expertos en matemáticas, trata sobre la codificación de números no enteros o "números en coma flotante". Un número decimal puede representarse en "notación científica" o "forma normalizada" como $N \times 10^n$, donde "N" es un número menor que 1Ø (que puede ser fraccionario), y "n" es un entero, positivo o negativo. Así, el número 1Ø2ØØ se expresa como $1.Ø2 \times 10^4$, y Ø.ØØØ345 se representa como $3.45 \times 1Ø^{-4}$. "N" se denomina "*mantisa*" *y* "*n*" *se llama* "*exponente*", según esta notación. Esta representación tiene la ventaja de que permite manejar más fácilmente los números muy grandes o muy pequeños.

Los números en coma flotante utilizan esta notación en base binaria. La mantisa ocupa cuatro octetos, y el exponente uno, pero los dígitos colocados en dichos octetos no son simplemente los que corresponden a los números en cuestión. Para empezar, el primer octeto de un número es el exponente más 128 (8ØH). Los cuatro octetos siguientes se destinan a la mantisa, que es una fracción binaria, cuyo valor decimal está entre Ø.5 y 1, sin llegar nunca a valer 1. El primer bit de la mantisa es siempre un 1, por lo que puede utilizarse como bit de signo. Si este bit se cambia por un Ø, entonces indica que el número es positivo; pero dejándolo a 1, señalará que el número es negativo. El exponente será 128 o mayor para números mayores o iguales que 1, y menor que 128 para los números fraccionarios sin parte entera.

El valor de la mantisa se representa como una fracción binaria. Del mismo modo que en los números binarios enteros utilizamos la posición para representar las potencias positivas de dos (1, 2, 4, 8, 16, 32, 64,...), podemos emplear las posiciones a la derecha del punto binario (¡no decimal!) para representar las potencias de dos negativas: 0.5, 0.25, 0.125, 0.0625,... que corresponden a 1/2, 1/4, 1/8, 1/16, etc. Por lo tanto, la fracción binaria .1Ø1ØØ1 representa: $0.5 + 0.125 + 0.015625$ $(1/2 + 1/8 + 1/64) = 0.640625$. Para pasar un número decimal a esta notación, se divide por la potencia de dos más cercana (que sea

superior a él). Por ejemplo, el número decimal 1.64 es mayor que 2^0 (que es 1), pero menor que 2^1 (que es 2); luego lo dividimos por 2 y lo escribimos así:

$$0.82\ (D) \times 2^1$$

2^1 nos da un exponente 1, que sumado a 128 resulta 129, el cual constituye el primer octeto de la codificación. 0.82 debe pasarse a una fracción binaria. Para esto, se compara a cada fracción de la serie binaria (0.5, 0.25, 0.125, etc.) y se resta sólo cuando la fracción binaria sea menor que el número. Es decir, empezando con el 0.82 y restando 0.5, nos queda 0.32, y el primer bit de la mantisa será un 1, como tiene que ocurrir siempre, si la conversión es correcta. 0.32 es mayor que la fracción binaria siguiente de la serie 0.25; por tanto, colocamos otro bit de la mantisa a 1 y restamos. El resultado es ahora 0.07, del que no podemos restar 0.125; en consecuencia, ponemos un Ø en la mantisa. Esta vez sí podemos realizar la sustración, porque el siguiente de la serie binaria es 0.0625, menor que 0.07. La operación se efectúa, sin olvidar añadir un nuevo 1 a la mantisa, quedándonos con 0.0075. Este proceso continuará hasta que en una resta obtengamos Ø como resultado, o hasta que hayamos calculado los 32 bits (cuatro octetos) de la mantisa. En realidad, la conversión se lleva a cabo para 33 bits, y si el que ocupa la posición 33 es un 1, entonces el bit número 32 se redondea a uno si era cero. El proceso de obtención de una fracción binaria no es *nunca exacto,* salvo para números que sean potencias negativas de dos, como 0.5, 0.125, etc., y por consiguiente, los números en coma flotante de los ordenadores y calculadoras tampoco serán nunca exactos.

Cuando se introduce el valor de una variable en la tabla de éstas, el computador tiene que llevar a cabo todo este trabajo, aunque, al menos, sólo lo hace una vez: cuando se introduce la línea. Si en su programa utiliza números, como los de la línea:

```
5Ø LET n = N*2.1416
```

la conversión de los códigos ASCII a un número en coma flotante ha de hacerse en el programa, y eso retrasará la ejecución de éste. Esta es la razón de que siempre se aconseje almacenar tales números en variables, al principio de los programas.

Apéndice C

Codificación de las tablas

Tablas numéricas

El primer octeto es el código ASCII + 32.

Los dos octetos siguientes son la longitud ocupada por la tabla desde el octeto siguiente a éstos, hasta el final.

El siguiente octeto es el número de dimensiones.

Los dos octetos siguientes forman la primera dimensión de la tabla.

A continuación van las dimensiones restantes, con dos octetos para cada una.

Por último, aparecen los elementos de la tabla (valores), cada uno de cinco octetos y ordenados: primero están todos los elementos cuyo primer subíndice es 1, ordenados a su vez; luego, los elementos con primer subíndice 2, y así sucesivamente, de forma que una configuración típica sería:

a(1,1), a(1,2), a(1,3), a(2,1), a(2,2), a(2,3), para una tabla de dimensiones 2 por 3.

Tablas de cadenas

Formalmente hablando, éstas son tablas de caracteres, reservando una dimensión para indicar el número de caracteres de cada elemento.

El primer octeto es el código ASCII + 96.

Los dos octetos siguientes contienen el número de octetos hasta el final de la tabla.

El octeto siguiente es el número de dimensiones.

Después vienen los octetos de dimensiones, dos para cada una. Una tabla de cadenas tiene que tener dos dimensiones como mínimo.

En último lugar se encuentran los elementos, pero sólo se necesita un octeto para cada uno, ya que dichos elementos son códigos de caracteres ASCII.

Apéndice D

Modos de direccionamiento del Z-8Ø

Existen algunas diferencias en los nombres utilizados para designar los modos de direccionamiento del Z-8Ø. Leventhal, por ejemplo, emplea en su libro la palabra "implícito" de manera distinta a la que aquí aparece, y con un sentido distinto del que tiene para otros autores.

Direccionamiento implícito: La dirección está implícita en la instrucción, y no es necesaria ninguna referencia a memoria. Ejemplo: RET.

Direccionamiento inmediato: La dirección del dato es la dirección de memoria siguiente a la dirección del octeto de la instrucción.

Direccionamiento directo: La dirección completa (dos octetos) de memoria está contenida en los dos octetos siguientes al octeto de la instrucción.

Direccionamiento indirecto por registro: La dirección del dato está contenida en un par de registros, normalmente HL.

Direccionamiento indexado: La dirección se obtiene sumando un octeto de desplazamiento (que forma parte del código) a una "dirección de base" contenida en un registro índice.

Direccionamiento relativo al programa: La dirección del dato se consigue añadiendo un octeto de desplazamiento, con su signo, a la dirección contenida en el contador de programa.

Direccionamiento de pila: El octeto está almacenado en la pila y se puede sacar de ésta con un comando POP, e introducir de nuevo con PUSH.

Direccionamiento en página cero: La instrucción RST es una forma especial de llamada a subrutina, que utiliza las direcciones de ØØH a 38H, en grupos de ocho. Se emplea mucho en el sistema operativo del Spectrum, y no está disponible para ningún otro fin.

Apéndice E

Tiempos de ejecución

Esta tabla presenta los tiempos en términos del número de pulsos del reloj, que necesitan las instrucciones para ejecutarse. En el caso del Spectrum, cuyo reloj funciona a 3.5 MHercios, el tiempo que dura un pulso de reloj es 0.2857142 μs. El microsegundo (μs) es una millonésima de segundo.

Operación	Tiempo	Comentarios
LD A,r	4	carga del registro A con cualquier otro
LD r,n	7	carga inmediata de cualquier registro
LD r,(HL)	7	carga indirecta de cualquier registro
LD A, (NN)	13	direccionamiento directo
LD RR,NN	10	carga de registro doble
LD HL,(NN)	16	carga de HL desde una posición de memoria
PUSH RR	11	meter un par de registros en la pila
POP RR	10	sacar un par de registros de la pila
ADD A,r	4	adición
INC r	4	incremento de un registro
INC(HL)	11	incrementar una posición de memoria
RLA	4	rotación del acumulador
JP NN	10	salto incondicional
JR desp	12	o salto incondicional relativo
JR condición	12	se cumple la condición (o sea, hay salto)
	7	condición falsa (no se salta)
CALL NN	17	llamada a la subrutina
RET	10	retorno de subrutina

Las instrucciones de desplazamiento y búsqueda de bloques no se han detallado aquí, porque su duración depende de cuántas veces se repitan las instrucciones. Se puede encontrar una lista completa de estos tiempos, en el libro de SGS-ATES, mencionado en el Apéndice A.

Apéndice F

Códigos de operación del Z-80: mnemónicos, códigos hexadecimales, decimales y binarios

A continuación, incluimos una lista de los 697 códigos de operación del Z-8Ø, que contiene los mnemónicos y los códigos en hexadecimal, decimal y en binario, dispuestos en cuatro columnas.

Cuando un código consta de dos o más octetos, éstos se han colocado verticalmente, para evitar confusiones. Donde son necesarios un octeto de datos, un desplazamiento o una dirección, aparecen como 00 para datos o desplazamientos, y como 0000 para las direcciones. Normalmente, esto se muestra en este tipo de listas con N para un octeto simple, y NN para una dirección (de ahí que el orden alfabético de esas listas no coincida con el nuestro, en lo que respecta a esas instrucciones); pero dado que para conseguir las listas se utilizó un programa conversor de hexadecimal a decimal y binario, las letras como N no pueden utilizarse en el programa.

El gran esfuerzo que supone la producción de listas como ésta, significa que, inevitablemente, se cometen algunas equivocaciones durante el proceso de recopilación de las mismas. He eliminado los errores que he podido descubrir, pero estaré muy agradecido si se pone en mi conocimiento cualquier equivocación que se encuentre.

ADC A,(HL)	8E	142	10001110
ADC A,(IX+00)	DD	221	11011101
	8E	142	10001110
	00	0	00000000
ADC A,(IY+00)	FD	253	11111101
	8E	142	10001110
	00	0	00000000
ADC A,A	8F	143	10001111
ADC A,B	88	136	10001000
ADC A,C	89	137	10001001
ADC A,D	8A	138	10001010
ADC A,00	CE	206	11001110
	00	0	00000000
ADC A,E	8B	139	10001011
ADC A,H	8C	140	10001100
ADC A,L	8D	141	10001101
ADC HL,BC	ED	237	11101101
	4A	74	01001010
ADC HL,DE	ED	237	11101101
	5A	90	01011010
ADC HL,HL	ED	237	11101101
	6A	106	01101010
ADC HL,SP	ED	237	11101101
	7A	122	01111010
ADD A,(HL)	86	134	10000110
ADD A,(IX+00)	DD	221	11011101
	86	134	10000110
	00	0	00000000
ADD A,(IY+00)	FD	253	11111101
	86	134	10000110
	00	0	00000000
ADD A,A	87	135	10000111
ADD A,B	80	128	10000000
ADD A,C	81	129	10000001
ADD A,D	82	130	10000010
ADD A,00	C6	198	11000110
	00	0	00000000
ADD A,E	83	131	10000011
ADD A,H	84	132	10000100
ADD A,L	85	133	10000101
ADD HL,BC	09	9	00001001
ADD HL,DE	19	25	00011001
ADD HL,HL	29	41	00101001
ADD HL,SP	39	57	00111001
ADD IX,BC	DD	221	11011101
	09	9	00001001
ADD IX,DE	DD	221	11011101
	19	25	00011001
ADD IX,IX	DD	221	11011101
	29	41	00101001
ADD IX,SP	DD	221	11011101
	39	57	00111001
ADD IY,BC	FD	253	11111101
	09	9	00001001
ADD IY,DE	FD	253	11111101
	19	25	00011001
ADD IY,IY	FD	253	11111101
	29	41	00101001
ADD IY,SP	FD	253	11111101
	39	57	00111001
AND (HL)	A6	166	10100110
AND (IX+00)	DD	221	11011101
	A6	166	10100110
	00	0	00000000

AND (IY+00)	FD	253	11111101
	A6	166	10100110
	00	0	00000000
AND A	A7	167	10100111
AND B	A0	160	10100000
AND C	A1	161	10100001
AND D	A2	162	10100010
AND 00	E6	230	11100110
	00	0	00000000
AND E	A3	163	10100011
AND H	A4	164	10100100
AND L	A5	165	10100101
BIT 0,(HL)	CB	203	11001011
	46	70	01000110
BIT 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	46	70	01000110
BIT 0,A	CB	203	11001011
	47	71	01000111
BIT 0,B	CB	203	11001011
	40	64	01000000
BIT 0,C	CB	203	11001011
	41	65	01000001
BIT 0,D	CB	203	11001011
	42	66	01000010
BIT 0,E	CB	203	11001011
	43	67	01000011
BIT 0,H	CB	203	11001011
	44	68	01000100
BIT 0,L	CB	203	11001011
	45	69	01000101
BIT 1,(HL)	CB	203	11001011
	4E	78	01001110
BIT 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	4E	78	01001110
BIT 1,A	CB	203	11001011
	4F	79	01001111
BIT 1,B	CB	203	11001011
	48	72	01001000
BIT 1,C	CB	203	11001011
	49	73	01001001
BIT 1,D	CB	203	11001011
	4A	74	01001010
BIT 1,E	CB	203	11001011
	4B	75	01001011
BIT 1,H	CB	203	11001011
	4C	76	01001100
BIT 1,L	CB	203	11001011
	4D	77	01001101
BIT 2,(HL)	CB	203	11001011
	56	86	01010110
BIT 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	56	86	01010110

BIT 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	56	86	01010110
BIT 2,A	CB	203	11001011
	57	87	01010111
BIT 2,B	CB	203	11001011
	50	80	01010000
BIT 2,C	CB	203	11001011
	51	81	01010001
BIT 2,D	CB	203	11001011
	52	82	01010010
BIT 2,E	CB	203	11001011
	53	83	01010011
BIT 2,H	CB	203	11001011
	54	84	01010100
BIT 2,L	CB	203	11001011
	55	85	01010101
BIT 3,(HL)	CB	203	11001011
	5E	94	01011110
BIT 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	5E	94	01011110
BIT 3,A	CB	203	11001011
	5F	95	01011111
BIT 3,B	CB	203	11001011
	58	88	01011000
BIT 3,C	CB	203	11001011
	59	89	01011001
BIT 3,D	CB	203	11001011
	5A	90	01011010
BIT 3,E	CB	203	11001011
	5B	91	01011011
BIT 3,H	CB	203	11001011
	5C	92	01011100
BIT 3,L	CB	203	11001011
	5D	93	01011101
BIT 4,(HL)	CB	203	11001011
	66	102	01100110
BIT 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	66	102	01100110
BIT 4,A	CB	203	11001011
	67	103	01100111
BIT 4,B	CB	203	11001011
	60	96	01100000
BIT 4,C	CB	203	11001011
	61	97	01100001
BIT 4,D	CB	203	11001011
	62	98	01100010
BIT 4,E	CB	203	11001011
	63	99	01100011
BIT 4,H	CB	203	11001011
	64	100	01100100
BIT 4,L	CB	203	11001011
	65	101	01100101
BIT 5,(HL)	CB	203	11001011
	6E	110	01101110
BIT 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	6E	110	01101110
BIT 5,A	CB	203	11001011
	6F	111	01101111
BIT 5,B	CB	203	11001011
	68	104	01101000
BIT 5,C	CB	203	11001011
	69	105	01101001
BIT 5,D	CB	203	11001011
	6A	106	01101010
BIT 5,E	CB	203	11001011
	6B	107	01101011
BIT 5,H	CB	203	11001011
	6C	108	01101100
BIT 5,L	CB	203	11001011
	6D	109	01101101
BIT 6,(HL)	CB	203	11001011
	76	118	01110110
BIT 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	76	118	01110110
BIT 6,A	CB	203	11001011
	77	119	01110111
BIT 6,B	CB	203	11001011
	70	112	01110000
BIT 6,C	CB	203	11001011
	71	113	01110001
BIT 6,D	CB	203	11001011
	72	114	01110010
BIT 6,E	CB	203	11001011
	73	115	01110011
BIT 6,H	CB	203	11001011
	74	116	01110100
BIT 6,L	CB	203	11001011
	75	117	01110101
BIT 7,(HL)	CB	203	11001011
	7E	126	01111110
BIT 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	7E	126	01111110
BIT 7,A	CB	203	11001011
	7F	127	01111111
BIT 7,B	CB	203	11001011
	78	120	01111000

BIT 7,C	CB	203	11001011
	79	121	01111001
BIT 7,D	CB	203	11001011
	7A	122	01111010
BIT 7,E	CB	203	11001011
	7B	123	01111011
BIT 7,H	CB	203	11001011
	7C	124	01111100
BIT 7,L	CB	203	11001011
	7D	125	01111101
CALL 00	CD	205	11001101
	00	0	00000000
CALL C,00	DC	220	11011100
	00	0	00000000
CALL M,00	FC	252	11111100
	00	0	00000000
CALL NC,00	D4	212	11010100
	00	0	00000000
CALL P,00	F4	244	11110100
	00	0	00000000
CALL PE,00	EC	236	11101100
	00	0	00000000
CALL PO,00	E4	228	11100100
	00	0	00000000
CALL Z,00	CC	204	11001100
	00	0	00000000
CCF	3F	63	00111111
CP(HL)	BE	190	10111110
CP(IX+0)	DD	221	11011101
	BE	190	10111110
	00	0	00000000
CP(IY+0)	FD	253	11111101
	BE	190	10111110
	00	0	00000000
CP A	BF	191	10111111
CP B	B8	184	10111000
CP C	B9	185	10111001
CP D	BA	186	10111010
CP 00	FE	254	11111110
	00	0	00000000
CP E	BB	187	10111011
CP H	BC	188	10111100
CP L	BD	189	10111101
CPD	ED	237	11101101
	A9	169	10101001
CPDR	ED	237	11101101
	B9	185	10111001
CPI	ED	237	11101101
	A1	161	10100001
CPIR	ED	237	11101101
	B1	177	10110001
CPL	2F	47	00101111
DAA	27	39	00100111
DEC(HL)	35	53	00110101
DEC(IX+0)	DD	221	11011101
	35	53	00110101
	00	0	00000000
DEC(IY+0)	FD	253	11111101
	35	53	00110101
	00	0	00000000
DEC A	3D	61	00111101
DEC B	05	5	00000101
DEC BC	0B	11	00001011
DEC C	0D	13	00001101
DEC D	15	21	00010101
DEC DE	1B	27	00011011
DEC E	1D	29	00011101
DEC H	25	37	00100101
DEC HL	2B	43	00101011
DEC IX	DD	221	11011101
	2B	43	00101011
DEC IY	FD	253	11111101
	2B	43	00101011
DEC L	2D	45	00101101
DEC SP	3B	59	00111011
DI	F3	243	11110011
DJNZ,00	10	16	00010000
	00	0	00000000
EI	FB	251	11111011
EX	8	8	10001000
EX(SP),HL	E3	227	11100011
EX(SP)IX	DD	221	11011101
	E3	227	11100011
EX(SP)IY	FD	253	11111101
	E3	227	11100011
EX AF,AF'	08	8	00001000
EX DE,HL	EB	235	11101011
EXX	D9	217	11011001
HLT	76	118	01110110
IM 0	ED	237	11101101
	46	70	01000110
IM 1	ED	237	11101101
	56	86	01010110
IM 2	ED	237	11101101
	5E	94	01011110
IN A,(C)	ED	237	11101101
	78	120	01111000
IN A,PORT	DB	219	11011011
	00	0	00000000
IN B,(C)	ED	237	11101101
	40	64	01000000
IN C,(C)	ED	237	11101101
	48	72	01001000
IN D,(C)	ED	237	11101101
	50	80	01010000
IN E,(C)	ED	237	11101101
	58	88	01011000
IN H,(C)	ED	237	11101101
	60	96	01100000
IN L,(C)	ED	237	11101101
	68	104	01101000
INC(HL)	34	52	00110100
INC(IX+00)	DD	221	11011101
	34	52	00110100
	00	0	00000000
INC(IY+00)	DD	221	11011101
	34	52	00110100
	00	0	00000000
INC A	3C	60	00111100
INC B	04	4	00000100
INC BC	03	3	00000011
INC C	0C	12	00001100
INC D	14	20	00010100
INC DE	13	19	00010011
INC E	1C	28	00011100
INC H	24	36	00100100
INC HL	23	35	00100011
INC IX	DD	221	11011101
	23	35	00100011

INC IY	FD	253	11111101
	23	35	00100011
INC L	2C	44	00101100
INC SP	33	51	00110011
IND	ED	237	11101101
	AA	170	10101010
INDR	ED	237	11101101
	BA	186	10111010
INI	ED	237	11101101
	A2	162	10100010
INIR	ED	237	11101101
	B2	178	10110010
JP(HL)	E9	233	11101001
JP(IX)	DD	221	11011101
	E9	233	11101001
JP(IY)	FD	253	11111101
	E9	233	11101001
J 0000	C3	195	11000011
	00	0	00000000
	00	0	00000000
JP C,0000	DA	218	11011010
	00	0	00000000
	00	0	00000000
JP M,0000	FA	250	11111010
	00	0	00000000
	00	0	00000000
JP NC,0000	D2	210	11010010
	00	0	00000000
	00	0	00000000
JP NZ,0000	C2	194	11000010
	00	0	00000000
	00	0	00000000
JP P,0000	F2	242	11110010
	00	0	00000000
	00	0	00000000
JP PE,0000	EA	234	11101010
	00	0	00000000
	00	0	00000000
JP PO,0000	E2	226	11100010
	00	0	00000000
	00	0	00000000
JP Z,0000	CA	202	11001010
	00	0	00000000
	00	0	00000000
JR C,00	38	56	00111000
	00	0	00000000
JR 00	18	24	00011000
	00	0	00000000
JR NC,00	30	48	00110000
	00	0	00000000
JR NZ,00	20	32	00100000
	00	0	00000000
JR Z,00	28	40	00101000
	00	0	00000000
LD(0000),A	32	50	00110010
	00	0	00000000
	00	0	00000000
LD(0000),BC	ED	237	11101101
	43	67	01000011
	00	0	00000000
	00	0	00000000
LD(0000),DE	ED	237	11101101
	53	83	01010011
	00	0	00000000
	00	0	00000000

LD(0000),HL	ED	237	11101101
	63	99	01100011
	00	0	00000000
	00	0	00000000
LD(0000),HL	22	34	00100010
LD(0000),IX	DD	221	11011101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),IY	FD	253	11111101
	22	34	00100010
	00	0	00000000
	00	0	00000000
LD(0000),SP	ED	237	11101101
	73	115	01110011
	00	0	00000000
	00	0	00000000
LD(BC),A	02	2	00000010
LD(DE),A	12	18	00010010
LD(HL),A	77	119	01110111
LD(HL),B	70	112	01110000
LD(HL),C	71	113	01110001
LD(HL),D	72	114	01110010
LD(HL),00	36	54	00110110
	00	0	00000000
LD(HL),E	73	115	01110011
LD(HL),H	74	116	01110100
LD(HL),L	75	117	01110101
LD(IX+00),A	DD	221	11011101
	77	119	01110111
	00	0	00000000
LD(IX+00),B	DD	221	11011101
	70	112	01110000
	00	0	00000000
LD(IX+00),C	DD	221	11011101
	71	113	01110001
	00	0	00000000
LD(IX+00),00	DD	221	11011101
	36	54	00110110
	00	0	00000000
	00	0	00000000
LD(IX+00),E	DD	221	11011101
	73	115	01110011
	00	0	00000000
LD(IX+00),H	DD	221	11011101
	74	116	01110100
	00	0	00000000
LD(IX+00),L	DD	221	11011101
	75	117	01110101
	00	0	00000000
LD(IY+00),A	FD	253	11111101
	77	119	01110111
	00	0	00000000
LD(IY+00),B	FD	253	11111101
	70	112	01110000
	00	0	00000000
LD(IY+00),C	FD	253	11111101
	71	113	01110001
	00	0	00000000
LD(IY+00),D	FD	253	11111101
	72	114	01110010
	00	0	00000000
LD(IY+00),00	FD	253	11111101
	36	54	00110110
	00	0	00000000
	00	0	00000000

LD(IY+00),E	FD	253	11111101
	73	115	01110011
	00	0	00000000
LD(IY+00),H	FD	253	11111101
	74	116	01110100
	00	0	00000000
LD(IY+00),L	FD	253	11111101
	75	117	01110101
	00	0	00000000
LD A,(0000)	3A	58	00111010
	00	0	00000000
	00	0	00000000
LD A,(BC)	0A	10	00001010
LD A,(DE)	1A	26	00011010
LD A,(HL)	7E	126	01111110
LD A,(IY+00)	DD	221	11011101
	7E	126	01111110
	00	0	00000000
LD A,(IX+00)	FD	253	11111101
	7E	126	01111110
	00	0	00000000
LD A,A	7F	127	01111111
LD A,B	78	120	01111000
LD A,C	79	121	01111001
LD A,D	7A	122	01111010
LD A,00	3E	62	00111110
	00	0	00000000
LD A,E	7B	123	01111011
LD A,H	7C	124	01111100
LD A,I	ED	237	11101101
	57	87	01010111
LD A,L	7D	125	01111101
LD A,R	ED	237	11101101
	5F	95	01011111
LD B,(HL)	46	70	01000110
LD B,(IX+00)	DD	221	11011101
	46	70	01000110
	00	0	00000000
LD B,(IY+00)	FD	253	11111101
	46	70	01000110
	00	0	00000000
LD B,A	47	71	01000111
LD B,B	40	64	01000000
LD B,C	41	65	01000001
LD B,D	42	66	01000010
LD B,00	06	6	00000110
	00	0	00000000
LD B,E	43	67	01000011
LD B,H	44	68	01000100
LD B,L	45	69	01000101
LD BC,(0000)	ED	237	11101101
	4B	75	01001011
	00	0	00000000
	00	0	00000000
LD BC,0000	01	1	00000001
	00	0	00000000
	00	0	00000000
LD C,(HL)	4E	78	01001110
LD C,(IX+00)	DD	221	11011101
	4E	78	01001110
	00	0	00000000
LD C,(IY+00)	FD	253	11111101
	4E	78	01001110
	00	0	00000000

LD C,A	4F	79	01001111
LD C,B	48	72	01001000
LD C,C	49	73	01001001
LD C,D	4A	74	01001010
LD C,00	0E	14	00001110
	00	0	00000000
LD C,E	4B	75	01001011
LD C,H	4C	76	01001100
LD C,L	4D	77	01001101
LD D,(HL)	56	86	01010110
LD D,(IX+00)	DD	221	11011101
	56	86	01010110
	00	0	00000000
LD D,(IY+00)	FD	253	11111101
	56	86	01010110
	00	0	00000000
LD D,A	57	87	01010111
LD D,B	50	80	01010000
LD D,C	51	81	01010001
LD D,D	52	82	01010010
LD D,00	16	22	00010110
	00	0	00000000
LD D,E	53	83	01010011
LD D,H	54	84	01010100
LD D,L	55	85	01010101
LD DE,(0000)	ED	237	11101101
	5B	91	01011011
	00	0	00000000
	00	0	00000000
LD DE,0000	11	17	00010001
	00	0	00000000
	00	0	00000000
LD E,(HL)	5E	94	01011110
LD E,(IX+00)	DD	221	11011101
	5E	94	01011110
	00	0	00000000
LD E,(IY+00	FD	253	11111101
	5E	94	01011110
	00	0	00000000
LD E,A	5F	95	01011111
LD E,B	58	88	01011000
LD E,C	59	89	01011001
LD E,D	5A	90	01011010
LD E,00	1E	30	00011110
	00	0	00000000
LD E,E	5B	91	01011011
LD E,H	5C	92	01011100
LD E,L	5D	93	01011101
LD H,(HL)	66	102	01100110
LD H,(IX+00)	DD	221	11011101
	66	102	01100110
	00	0	00000000
LD H,(IY+00)	FD	253	11111101
	66	102	01100110
	00	0	00000000
LD H,A	67	103	01100111
LD H,B	60	96	01100000
LD H,C	61	97	01100001
LD H,D	62	98	01100010
LD H,00	26	38	00100110
	00	0	00000000
LD H,E	63	99	01100011
LD H,H	64	100	01100100
LD H,L	65	101	01100101

LD HL,0000	ED	237	11101101
	6B	107	01101011
	00	0	00000000
	00	0	00000000
LD HL,(0000)	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD HL,0000	21	33	00100001
	00	0	00000000
	00	0	00000000
LD I,A	ED	237	11101101
	47	71	01000111
LD IX,(0000)	DD	221	11011101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IX,0000	DD	221	11011101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD IY,(0000)	FD	253	11111101
	2A	42	00101010
	00	0	00000000
	00	0	00000000
LD IY,0000	FD	253	11111101
	21	33	00100001
	00	0	00000000
	00	0	00000000
LD L,(HL)	6E	110	01101110
LD L,(IX+00)	DD	221	11011101
	6E	110	01101110
	00	0	00000000
LD L,(IY+00)	FD	253	11111101
	6E	110	01101110
	00	0	00000000
LD L,A	6F	111	01101111
LD L,B	68	104	01101000
LD L,C	69	105	01101001
LD L,D	6A	106	01101010
LD L,00	2E	46	00101110
	00	0	00000000
LD L,E	6B	107	01101011
LD L,H	6C	108	01101100
LD L,I	6D	109	01101101
LD R,A	ED	237	11101101
	4F	79	01001111
LD SP,(0000)	ED	237	11101101
	7B	123	01111011
	00	0	00000000
	00	0	00000000
LD SP,0000	31	49	00110001
	00	0	00000000
	00	0	00000000
LD SP,HL	F9	249	11111001
LD SP,IX	DD	221	11011101
	F9	249	11111001
LD SP,IY	FD	253	11111101
	F9	249	11111001
LDD	ED	237	11101101
	A8	168	10101000
LDDR	ED	237	11101101
	B8	184	10111000
LDI	ED	237	11101101
	A0	160	10100000

LDIR	ED	237	11101101
	B0	176	10110000
NEG	ED	237	11101101
	44	68	01000100
NOP	00	0	00000000
OR(HL)	B6	182	10110110
OR(IX+00)	DD	221	11011101
	B6	182	10110110
	00	0	00000000
OR(IY+00)	FD	253	11111101
	B6	182	10110110
	00	0	00000000
OR A	B7	183	10110111
OR B	B0	176	10110000
OR C	B1	177	10110001
OR D	B2	178	10110010
OR 00	F6	246	11110110
	00	0	00000000
OR E	B3	179	10110011
OR H	B4	180	10110100
OR L	B5	181	10110101
OTDR	ED	237	11101101
	BB	187	10111011
OTIR	ED	237	11101101
	B3	179	10110011
OUT(C),A	ED	237	11101101
	79	121	01111001
OUT(C),B	ED	237	11101101
	41	65	01000001
OUT(C),C	ED	237	11101101
	49	73	01001001
OUT(C),D	ED	237	11101101
	51	81	01010001
OUT(C),E	ED	237	11101101
	59	89	01011001
OUT(C),H	ED	237	11101101
	61	97	01100001
OUT(C),L	ED	237	11101101
	69	105	01101001
OUT(Port),A	D3	211	11010011
	00	0	00000000
OUTD	ED	237	11101101
	AB	171	10101011
OUTI	ED	237	11101101
	A3	163	10100011
POP AF	F1	241	11110001
POP BC	C1	193	11000001
POP DE	D1	209	11010001
POP HL	E1	225	11100001
POP IX	DD	221	11011101
	E1	225	11100001
POP IY	FD	253	11111101
	E1	225	11100001
PUSH AF	F5	245	11110101
PUSH BC	C5	197	11000101
PUSH DE	D5	213	11010101
PUSH HL	E5	229	11100101
PUSH IX	DD	221	11011101
	E5	229	11100101
PUSH IY	FD	253	11111101
	E5	229	11100101
RES 0,(HL)	CB	203	11001011
	86	134	10000110
RES 0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	00	0	00000000

RES 0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 0,A	CB	203	11001011
	87	135	10000111
RES 0,B	CB	203	11001011
	80	128	10000000
RES 0,C	CB	203	11001011
	81	129	10000001
RES 0,D	CB	203	11001011
	82	130	10000010
RES 0,E	CB	203	11001011
	83	131	10000011
RES 0,H	CB	203	11001011
	84	132	10000100
RES 0,L	CB	203	11001011
	85	133	10000101
RES 1,(HL)	CB	203	11001011
	8E	142	10001110
RES 1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	00	0	00000000
RES 1,A	CB	203	11001011
	8F	143	10001111
RES 1,B	CB	203	11001011
	88	136	10001000
RES 1,C	CB	203	11001011
	89	137	10001001
RES 1,D	CB	203	11001011
	8A	138	10001010
RES 1,E	CB	203	11001011
	8B	139	10001011
RES 1,H	CB	203	11001011
	8C	140	10001100
RES 1,L	CB	203	11001011
	8D	141	10001101
RES 2,(HL)	CB	203	11001011
	96	150	10010110
RES 2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	96	150	10010110
RES 2,A	CB	203	11001011
	97	151	10010111
RES 2,B	CB	203	11001011
	90	144	10010000
RES 2,C	CB	203	11001011
	91	145	10010001
RES 2,D	CB	203	11001011
	92	146	10010010
RES 2,E	CB	203	11001011
	93	147	10010011
RES 2,H	CB	203	11001011
	94	148	10010100
RES 2,L	CB	203	11001011
	95	149	10010101
RES 3,(HL)	CB	203	11001011
	9E	158	10011110
RES 3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	9E	158	10011110
RES 3,A	CB	203	11001011
	9F	159	10011111
RES 3,B	CB	203	11001011
	98	152	10011000
RES 3,C	CB	203	11001011
	99	153	10011001
RES 3,D	CB	203	11001011
	9A	154	10011010
RES 3,E	CB	203	11001011
	9B	155	10011011
RES 3,H	CB	203	11001011
	9C	156	10011100
RES 3,L	CB	203	11001011
	9D	157	10011101
RES 4,(HL)	CB	203	11001011
	A6	166	10100110
RES 4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	A6	166	10100110
RES 4,A	CB	203	11001011
	A7	167	10100111
RES 4,B	CB	203	11001011
	A0	160	10100000
RES 4,C	CB	203	11001011
	A1	161	10100001
RES 4,D	CB	203	11001011
	A2	162	10100010
RES 4,E	CB	203	11001011
	A3	163	10100011
RES 4,H	CB	203	11001011
	A4	164	10100100
RES 4,L	CB	203	11001011
	A5	165	10100101
RES 5,(HL)	CB	203	11001011
	AE	174	10101110
RES 5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	AE	174	10101110
RES 5,A	CB	203	11001011
	5F	95	01011111
RES 5,B	CB	203	11001011
	A8	168	10101000

RES 5,C	CB	203	11001011
	A9	169	10101001
RES 5,D	CB	203	11001011
	AA	170	10101010
RES 5,E	CB	203	11001011
	AB	171	10101011
RES 5,H	CB	203	11001011
	AC	172	10101100
RES 5,L	CB	203	11001011
	AD	173	10101101
RES 6,(HL)	CB	203	11001011
	B6	182	10110110
RES 6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	B6	182	10110110
RES 6,A	CB	203	11001011
	B7	183	10110111
RES 6,B	CB	203	11001011
	B0	176	10110000
RES 6,C	CB	203	11001011
	B1	177	10110001
RES 6,D	CB	203	11001011
	B2	178	10110010
RES 6,E	CB	203	11001011
	B3	179	10110011
RES 6,H	CB	203	11001011
	B4	180	10110100
RES 6,L	CB	203	11001011
	B5	181	10110101
RES 7,(HL)	CB	203	11001011
	BE	190	10111110
RES 7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	BE	190	10111110
RES 7,A	CB	203	11001011
	BF	191	10111111
RES 7,B	CB	203	11001011
	B8	184	10111000
RES 7,C	CB	203	11001011
	B9	185	10111001
RES 7,D	CB	203	11001011
	BA	186	10111010
RES 7,E	CB	203	11001011
	BB	187	10111011
RES 7,H	CB	203	11001011
	BC	188	10111100
RES 7,L	CB	203	11001011
	BD	189	10111101
RET	C9	201	11001001
RET C	D8	216	11011000
RET M	F8	248	11111000
RET NC	D0	208	11010000
RET NZ	C0	192	11000000
RET P	F0	240	11110000

RET PE	E8	232	11101000
RET PO	E0	224	11100000
RET Z	C8	200	11001000
RETI	ED	237	11101101
	4D	77	01001101
RETN	ED	237	11101101
	45	69	01000101
RL(HL)	CB	203	11001011
	16	22	00010110
RL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	16	22	00010110
RL A	CB	203	11001011
	17	23	00010111
RL B	CB	203	11001011
	10	16	00010000
RL C	CB	203	11001011
	11	17	00010001
RL D	CB	203	11001011
	12	18	00010010
RL E	CB	203	11001011
	13	19	00010011
RL H	CB	203	11001011
	14	20	00010100
RL L	CB	203	11001011
	15	21	00010101
RLA	17	23	00010111
RLC(HL)	CB	203	11001011
	06	6	00000110
RLC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	06	6	00000110
RLC A	CB	203	11001011
	07	7	00000111
RLC B	CB	203	11001011
	00	0	00000000
RLC C	CB	203	11001011
	01	1	00000001
RLC D	CB	203	11001011
	02	2	00000010
RLC E	CB	203	11001011
	03	3	00000011
RLC H	CB	203	11001011
	04	4	00000100
RLC L	CB	203	11001011
	05	5	00000101
RLCA	07	7	00000111
RLD	ED	237	11101101
	6F	111	01101111
RR(HL)	CB	203	11001011
	1E	30	00011110
RR(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110

RR(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	1E	30	00011110
RR A	CB	203	11001011
	1F	31	00011111
RR B	CB	203	11001011
	18	24	00011000
RR C	CB	203	11001011
	19	25	00011001
RR D	CB	203	11001011
	1A	26	00011010
RR E	CB	203	11001011
	1B	27	00011011
RR H	CB	203	11001011
	1C	28	00011100
RR L	CB	203	11001011
	1D	29	00011101
RRA	1F	31	00011111
RRC(HL)	CB	203	11001011
	0E	14	00001110
RRC(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	0E	14	00001110
RRC A	CB	203	11001011
	0F	15	00001111
RRC B	CB	203	11001011
	08	8	00001000
RRC C	CB	203	11001011
	09	9	00001001
RRC D	CB	203	11001011
	0A	10	00001010
RRC E	CB	203	11001011
	0B	11	00001011
RRC H	CB	203	11001011
	0C	12	00001100
RRC L	CB	203	11001011
	0D	13	00001101
RRCA	0F	15	00001111
RRD	ED	237	11101101
	67	103	01100111
RST 00	C7	199	11000111
RST 08	CF	207	11001111
RST 10	D7	215	11010111
RST 18	DF	223	11011111
RST 20	E7	231	11100111
RST 28	EF	239	11101111
RST 30	F7	247	11110111
RST 38	FF	255	11111111
SBC A,(HL)	9E	158	10011110
SBC A,(IX+00)	DD	221	11011101
	9E	158	10011110
	00	0	00000000
SBC A,(IY+00)	FD	253	11111101
	9E	158	10011110
	00	0	00000000
SBC A,A	9F	159	10011111
SBC A,B	98	152	10011000
SBC A,C	99	153	10011001
SBC A,D	9A	154	10011010
SBC A,00	DE	222	11011110
	00	0	00000000
SBC A,E	9B	155	10011011
SBC A,H	9C	156	10011100
SBC A,L	9D	157	10011101
SBC HL,BC	ED	237	11101101
	42	66	01000010
SBC HL,DE	ED	237	11101101
	52	82	01010010
SBC HL,HL	ED	237	11101101
	62	98	01100010
SBC HL,SP	ED	237	11101101
	72	114	01110010
SCF	37	55	00110111
SET 0,(HL)	CB	203	11001011
	C6	198	11000110
SET0,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	C6	198	11000110
SET0,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	23	35	00100011
SET0,A	CB	203	11001011
	C7	199	11000111
SET0,B	CB	203	11001011
	C0	192	11000000
SET0,C	CB	203	11001011
	C1	193	11000001
SET0,D	CB	203	11001011
	C2	194	11000010
SET0,E	CB	203	11001011
	C3	195	11000011
SET0,H	CB	203	11001011
	C4	196	11000100
SET0,L	CB	203	11001011
	C5	197	11000101
SET1,(HL)	CB	203	11001011
	CE	206	11001110
SET1,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110
SET1,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	CE	206	11001110
SET1,A	CB	203	11001011
	CF	207	11001111
SET1,B	CB	203	11001011
	C8	200	11001000
SET1,C	CB	203	11001011
	C9	201	11001001
SET1,D	CB	203	11001011
	CA	202	11001010
SET1,E	CB	203	11001011
	CB	203	11001011
SET1,H	CB	203	11001011
	CC	204	11001100
SET1,L	CB	203	11001011
	CD	205	11001101
SET2,(HL)	CB	203	11001011
	D6	214	11010110

SET2,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	D6	214	11010110
SET2,A	CB	203	11001011
	D7	215	11010111
SET2,B	CB	203	11001011
	D0	208	11010000
SET2,C	CB	203	11001011
	D1	209	11010001
SET2,D	CB	203	11001011
	D2	210	11010010
SET2,E	CB	203	11001011
	D3	211	11010011
SET2,H	CB	203	11001011
	D4	212	11010100
SET2,L	CB	203	11001011
	D5	213	11010101
SET3,(HL)	CB	203	11001011
	DE	222	11011110
SET3,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	DE	222	11011110
SET3,A	CB	203	11001011
	DF	223	11011111
SET3,B	CB	203	11001011
	D8	216	11011000
SET3,C	CB	203	11001011
	D9	217	11011001
SET3,D	CB	203	11001011
	DA	218	11011010
SET3,E	CB	203	11001011
	DB	219	11011011
SET3,H	CB	203	11001011
	DC	220	11011100
SET3,L	CB	203	11001011
	DD	221	11011101
SET4,(HL)	CB	203	11001011
	E6	230	11100110
SET4,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	E6	230	11100110
SET4,A	CB	203	11001011
	E7	231	11100111
SET4,B	CB	203	11001011
	E0	224	11100000
SET4,C	CB	203	11001011
	E1	225	11100001
SET4,D	CB	203	11001011
	E2	226	11100010
SET4,E	CB	203	11001011
	E3	227	11100011
SET4,H	CB	203	11001011
	E4	228	11100100
SET4,L	CB	203	11001011
	E5	229	11100101
SET5,(HL)	CB	203	11001011
	EE	238	11101110
SET5,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	EE	238	11101110
SET5,A	CB	203	11001011
	EF	239	11101111
SET5,B	CB	203	11001011
	E8	232	11101000
SET5,C	CB	203	11001011
	E9	233	11101001
SET5,D	CB	203	11001011
	EA	234	11101010
SET5,E	CB	203	11001011
	EB	235	11101011
SET5,H	CB	203	11001011
	EC	236	11101100
SET5,L	CB	203	11001011
	ED	237	11101101
SET6,(HL)	CB	203	11001011
	F6	246	11110110
SET6,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	F6	246	11110110
SET6,A	CB	203	11001011
	F7	247	11110111
SET6,B	CB	203	11001011
	F0	240	11110000
SET6,C	CB	203	11001011
	F1	241	11110001
SET6,D	CB	203	11001011
	F2	242	11110010
SET6,E	CB	203	11001011
	F3	243	11110011
SET6,H	CB	203	11001011
	F4	244	11110100
SET6,L	CB	203	11001011
	F5	245	11110101
SET7,(HL)	CB	203	11001011
	FE	254	11111110
SET7,(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110
SET7,(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	FE	254	11111110

SET7,A	CB	203	11001011
	FF	255	11111111
SET7,B	CB	203	11001011
	F8	248	11111000
SET7,C	CB	203	11001011
	F9	249	11111001
SET7,D	CB	203	11001011
	FA	250	11111010
SET7,E	CB	203	11001011
	FB	251	11111011
SET7,H	CB	203	11001011
	FC	252	11111100
SET7,L	CB	203	11001011
	FD	253	11111101
SLA(HL)	CB	203	11001011
	26	38	00100110
SLA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	26	38	00100110
SLA A	CB	203	11001011
	27	39	00100111
SLA B	CB	203	11001011
	20	32	00100000
SLA C	CB	203	11001011
	21	33	00100001
SLA D	CB	203	11001011
	22	34	00100010
SLA E	CB	203	11001011
	23	35	00100011
SLA H	CB	203	11001011
	24	36	00100100
SLA L	CB	203	11001011
	25	37	00100101
SRA(HL)	CB	203	11001011
	2E	46	00101110
SRA(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	2E	46	00101110
SRA A	CB	203	11001011
	2F	47	00101111
SRA B	CB	203	11001011
	28	40	00101000
SRA C	CB	203	11001011
	29	41	00101001
SRA D	CB	203	11001011
	2A	42	00101010
SRA E	CB	203	11001011
	2B	43	00101011
SRA H	CB	203	11001011
	2C	44	00101100
SRA L	CB	203	11001011
	2D	45	00101101
SRL(HL)	CB	203	11001011
	3E	62	00111110
SRL(IX+00)	DD	221	11011101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL(IY+00)	FD	253	11111101
	CB	203	11001011
	00	0	00000000
	3E	62	00111110
SRL A	CB	203	11001011
	3F	63	00111111
SRL B	CB	203	11001011
	38	56	00111000
SRL C	CB	203	11001011
	39	57	00111001
SRL D	CB	203	11001011
	3A	58	00111010
SRL E	CB	203	11001011
	3B	59	00111011
SRL H	CB	203	11001011
	3C	60	00111100
SRL L	CB	203	11001011
	3D	61	00111101
SUB(HL)	96	150	10010110
SUB(IX+00)	DD	221	11011101
	96	150	10010110
	00	0	00000000
SUB(IY+00)	FD	253	11111101
	96	150	10010110
	00	0	00000000
SUB A	97	151	10010111
SUB B	90	144	10010000
SUB C	91	145	10010001
SUB D	92	146	10010010
SUB 00	D6	214	11010110
	00	0	00000000
SUB E	93	147	10010011
SUB H	94	148	10010100
SUB L	95	149	10010101
XOR(HL)	AE	174	10101110
XOR(IX+00)	DD	221	11011101
	AE	174	10101110
	00	0	00000000
XOR(IY+00)	FD	253	11111101
	AE	174	10101110
	00	0	00000000
XOR A	AF	175	10101111
XOR B	A8	168	10101000
XOR C	A9	169	10101001
XOR D	AA	170	10101010
XOR 00	EE	238	11101110
	00	0	00000000
XOR E	AB	171	10101011
XOR H	AC	172	10101100
XOR L	AD	173	10101101

Apéndice G

Códigos ASCII

SPACE	20	32	00100000
!	21	33	00100001
"	22	34	00100010
£	23	35	00100011
$	24	36	00100100
%	25	37	00100101
&	26	38	00100110
'	27	39	00100111
(	28	40	00101000
)	29	41	00101001
*	2A	42	00101010
+	2B	43	00101011
,	2C	44	00101100
–	2D	45	00101101
.	2E	46	00101110
/	2F	47	00101111
0	30	48	00110000
1	31	49	00110001
2	32	50	00110010
3	33	51	00110011
4	34	52	00110100
5	35	53	00110101
6	36	54	00110110
7	37	55	00110111
8	38	56	00111000
9	39	57	00111001
:	3A	58	00111010
;	3B	59	00111011
<	3C	60	00111100
=	3D	61	00111101
>	3E	62	00111110
?	3F	63	00111111
@	40	64	01000000
A	41	65	01000001
B	42	66	01000010
C	43	67	01000011
D	44	68	01000100
E	45	69	01000101
F	46	70	01000110
G	47	71	01000111
H	48	72	01001000
I	49	73	01001001
J	4A	74	01001010
K	4B	75	01001011
L	4C	76	01001100
M	4D	77	01001101
N	4E	78	01001110
O	4F	79	01001111

P	50	80	01010000
Q	51	81	01010001
R	52	82	01010010
S	53	83	01010011
T	54	84	01010100
U	55	85	01010101
V	56	86	01010110
W	57	87	01010111
X	58	88	01011000
Y	59	89	01011001
Z	5A	90	01011010
[	5B	91	01011011
/	5C	92	01011100
]	5D	93	01011101
^	5E	94	01011110
_	5F	95	01011111
`	60	96	01100000
a	61	97	01100001
b	62	98	01100010
c	63	99	01100011
d	64	100	01100100
e	65	101	01100101
f	66	102	01100110
g	67	103	01100111
h	68	104	01101000
i	69	105	01101001
j	6A	106	01101010
k	6B	107	01101011
l	6C	108	01101100
m	6D	109	01101101
n	6E	110	01101110
o	6F	111	01101111
p	70	112	01110000
q	71	113	01110001
r	72	114	01110010
s	73	115	01110011
t	74	116	01110100
u	75	117	01110101
v	76	118	01110110
w	77	119	01110111
x	78	120	01111000
y	79	121	01111001
z	7A	122	01111010
{	7B	123	01111011
|	7C	124	01111100
}	7D	125	01111101
~	7E	126	01111110
©	7F	127	01111111

Indice

Acumulador, 50, 52
Almacenamiento de enteros, 22
AND, 10
Aritmética, 9 (operaciones aritméticas)
Asignación dinámica, 24 (asignada dinámicamente)

Base decimal, 6, 35
BASIC, 1
Baudios, 106
Binario, 35
Bit, 1
Bit de altavoz, 10
Bit de arranque, 106
Bit de parada, 106
Bloque de destino, 120
Bloque fuente, 120
Bloqueo del computador, 69, 109
Borrado de memoria, 74
BREAK, 34
Bucle FOR...NEXT, 24
Bucle infinito, 34, 111
Bucles, 88
Bucles erróneos, 111
Bus de direcciones, 47
Byte, 3
Byte de instrucción, 34
Bytes de longitud de línea, 26

Cabecera de línea, 27
Campbell, 112
Carga, 8
Carga del código máquina, 92
CLEAR, 68
Codificación de tablas de cadena, 140
Codificación de tablas numéricas, 140
Código ASCII, 5
Código binario, 3
Código fuente, 113
Código hexadecimal, 35, 36
Código independiente de la posición, 94
Código máquina, 5, 12, 35, 66
Código objeto, 113
Códigos, 4
Códigos de operación del Z-8Ø, 143
Códigos numéricos, 35
Color del borde, 100
Comando de intercambio, 125
Comandos de los puertos de E/S, 95
Comentarios del programa, 63
Comparación, 62
Complemento á 2, 45
Contador de programa, 47
Conversión decimal-hexadecimal, 39
Conversión hexadecimal-decimal, 39
Cristal de cuarzo, 90

Decisión, 77
Declaración de variables, 17
Decremento, 62, 120
Depuración, 108
Desensamblador Infrared, 36, 114
Desensamblaje comentado, 87
Desplazamiento, 54, 55
Diagrama de bloques, 7
Dígito binario, 1
Dirección, 5
Dirección de comienzo de BASIC, 25
Dirección de página, 59
Dirección de puerto, 96, 98

Dirección del número de línea, 131
Direccionamiento directo, 51
Direccionamiento indirecto, 52
Direccionamiento indirecto por registro, 52
Direccionamiento inmediato, 50
Direccionamiento relativo al PC, 54
Direcciones importantes, 17
Dispositivo programable, 10

Efectos visuales, 105
Ejecución de programa, 28
Ensamblador, 36, 49, 113
Ensamblador DPAS, 36
Ensamblador ULTRAVIOLET, 108, 113
Ensamblador ZEN, 117
Entero negativo, 22
Entrada/Salida, 77
Errores de redondeo, 21
Errores en ensamblador, 116
Escritura, 12
Exponente, 138

Falso, 86
Fichero de pantalla, 29, 30, 121
FORTH, 129

Gnerador de pulsos de reloj, 32
Grabación del código máquina, 92

Hardware, 33

Impresora en serie, 105
Incremento, 48, 62, 120
Indicador C, 56
Indicador S, 56
Indicador Z, 56
Indicadores, 56
Indice, 17
Información inválida, 17
Inicialización, 16, 17
Instrucción CALL, 85
Instrucción CP, 64
Instrucción DJNZ, 88
Instrucción JP, 66
Instrucción JR, 66
Instrucción LDIR, 119
Instrucción POP, 69, 70
Instrucción PUSH, 69, 70
Instrucción RET, 85
Instrucción RLA, 75
Instrucciones aritméticas, 61
Instrucciones de búsqueda de bloques, 123
Instrucciones de desplazamiento, 61, 119
Instrucciones de salto, 67
Instrucciones de salto relativo, 57
Instrucciones lógicas, 9
Interruptor, 1

Juego de instrucciones, 40

Lectura, 12
Libros, 137
Línea multisentencia, 27

Mantisa, 138
Memoria, 1
Mensajes de error, 34, 125
Microprocesador, 7, 32
Mnemónicos, 49
Modos de direccionamiento, 48, 141
Monitores, 112

Notación científica, 138
Número binario, 35
Número con signo, 43
Número sin signo, 43
Números de línea, 19
Números en coma flotante, 138
Números negativos, 43

Obtención de un carácter, 78
Octeto, 3
Operador, 50
Operando, 50
OR, 9
ORG, 70
Organigramas, 77

Palabras clave, 6, 30
Paréntesis, 52
Paridad, 107
Paso de valores, 122
Paso de variables, 129
PEEK, 5
Personal Computer World, 137
Peso de un dígito, 3
Pila, 69
Píns de datos, 12

POKE, 19
PRINT, 5
Problemas con los programas, 113
Proceso, 77
Programa: buscador de líneas, 133, 136
  cambio a mayúsculas, 83
  conversión numérica, 43, 44
  teclas pulsadas, 97
Programas prácticos, 70
Pseudoinstrucciones, 116
Puerto, 14
Puerto de salida, 95
Punto de parada, 110
Punto y coma, 63

RAM, 4
Referencias adelantadas, 116
Registro contador de programa (PC), 47
Registro de estado, 56
Registro de indicadores, 56
Registro de interrupción, 56
Registro de refresco, 56
Registros, 49
Registros alternativos, 60
Registros índice, 58
Registros simples, 56
Reglas lógicas, 9, 11
Reinicialización por hardware, 111
Resta, 9
Retardos de tiempo, 90
Revistas, 137
ROM, 4
Rotación, 61
RS-232, 106
RST 8, 125
Rutina de PRINT, 29
Rutinas auxiliares, 30
Rutinas de la ROM, 87
Rutinas numéricas, 127

Salto condicional, 57
Señales con dos líneas, 2
Señales de lectura, 48
Software, 33
Sonidos: «click», 102
  «zumbido», 104
Subrutinas, 16
Suma, 9

Tabla de palabras clave, 7
Tabla de variables, 17, 122
Tampón, 24
Terminador, 77
Tiempos de las operaciones, 142
Tipos de instrucciones, 34
TOKEN, 5
Tomas de restas, 83
Transferencias de bytes, 8

UCP, 7
Ultima línea del programa, 19
USR, 68

Variable BASIC, 122
Variable de cadena, 23
Variable numérica, 20
Velocidad de cuenta, 89
Velocidad del reloj, 32
VERIFY, 95
Vuelta de subrutina, 69

XOR, 9

Z-8Ø, 8, 32, 47

Spectrum 48k

OTROS TITULOS SPECTRUM

ZX SPECTRUM. COMO OBTENER EL MAXIMO RENDIMIENTO
Sinclair, I.

SPECTRUM. LIBRO DE JUEGOS
James, M.

EL PROGRAMADOR DE SPECTRUM
Gee, S. N.

CUARENTA JUEGOS EDUCATIVOS PARA EL SPECTRUM
Apps, V.

DIAZ DE SANTOS, S.A.

MADRID/BARCELONA